Tampereen yliopisto

Jermu Mäki

WEB-OHJELMISTON UUDISTAMINEN FUNKTIONAALISELLA PARADIGMALLA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Lokakuu 2019

TIIVISTELMÄ

Jermu Mäki: Web-ohjelmiston uudistaminen funktionaalisella paradigmalla
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Lokakuu 2019

Ohjelmistot vanhenevat ajan saatossa. Vanheneminen johtuu ympäristön muutoksista. Ohjelmistoja voidaan ylläpitää, mutta jossain vaiheessa muuttuneet vaatimukset poikkeavat niin paljon alkuperäisistä tai muutosten tekeminen on niin hankalaa, että ohjelmiston uudistaminen on kannattavampaa. Uudistus toimii investointina, jonka hyöty realisoituu myöhemmässä vaiheessa. Uudistamiseen voidaan päätyä lisäksi paremman joustavuuden ja ylläpidon saavuttamiseksi.

Ohjelmiston uudistaminen voidaan jaotella sen vaikutusten mukaan. Pienimmillään uudistus koskettaa vain tiettyä osakokonaisuutta kuten käyttöliittymää ja suurimmillaan uudistus voi koskea koko järjestelmää. Ohjelmien uudistamisen tekeminen pohjautuu uudistettavan ohjelman tutkimiseen eri tasoilla. Alkuperäisen ohjelman lähdekoodi ja suunnitteludokumentit tukevat uudelleensuunnittelua. Suunnittelun lopputuloksena syntyy uusi korkeamman tason arkkitehtuurimalli, jota aletaan toteuttaa.

Tässä työssä tutkitaan laajan web-järjestelmän erään osakokonaisuuden uudistamista. Toiminnalliset vaatimukset ovat muuttuneet alkuperäisistä ohjelman elinkaaren aikana. Ylläpito ja uusien muutosten tekeminen ovat osoittautuneet hankaliksi. Uudistus tehdään korvaamalla vanha järjestelmä uusien määrittelyiden perusteella, mutta myös tutkimalla vanhan järjestelmän lähdekoodia. Vanhan järjestelmän oliopohjainen Java-kieli vaihdetaan funktionaalista paradigmaa noudattavaan ja ilmaisuvoimaisempaan Clojureen.

Ohjelman arkkitehtuuri suunniteltiin uudelleen ja ohjelmointikieli vaihdettiin paremmin datankäsittelyyn soveltuvaksi ja yleiskäyttöisemmäksi. Näillä muutoksilla saavutettiin yksinkertaisempi ohjelman tekninen rakenne ja huomattava vähennys lähdekoodin rivimäärässä. Ohjelmistometriikoin, kyselyn ja haastattelujen perusteella arvioituna uudistettu ohjelma oli erityisesti ylläpidon ja suorituskyvyn kannalta onnistunut. Funktionaalinen ohjelmointikieli koettiin sopivan datankäsittelyä sisältävän web-järjestelmän tekemiseen.

Lopputuloksena oli selkeästi parempi järjestelmä. Tärkeimmät tavoitteet olivat parempi ylläpidettävyys, mahdollisuus korjata virheellistä dataa ja parempi suorituskyky. Näihin tavoitteisiin päästiin. Vanhan datan migraatio uuteen järjestelmään oli työlästä ja se vei enemmän aikaa kuin alun perin ennakoitiin. Kaiken kaikkiaan uudistus oli onnistunut ja funktionaalinen ohjelmointiparadigma sopii web-ohjelmistojen toteutukseen ja runsaasti ohjelmallista datankäsittelyä sisältäviin ohjelmiin. Tärkeää on kuitenkin myös ohjelmistosuunnittelu, laadukas ohjelmakoodi ja hyvät ohjelmointikäytännöt.

Avainsanat: ohjelmistojen uudistaminen, funktionaalinen ohjelmointi, web-ohjelmisto

ABSTRACT

Jermu Mäki: Re-engineering web-based software using functional paradigm
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
October 2019

Computer software will age and deteriorate as time passes by. Aging is caused by continuously changing environment. Software can be maintained but eventually requirements differ too much from the originals or implementing new changes is too difficult that re-engineering the software is more cost-effective. Re-engineering is an investment that will pay-off in future. Re-engineering can also be started in order to achieve better flexibility or maintainability.

Software re-engineering can be categorized by size of its effects. The smallest re-engineering work affects only a specific part of system like user interface and biggest one is a system wide replacement. Re-engineering is based on investigation of an old system in different levels. Original source code and design documents support re-engineering design process. The result is a new high-level architecture model which will be implemented.

This thesis contains investigation of re-engineering of large web-based system. The re-engineering covers one logical subsystem. The functional requirements have changed from the original ones. Also, maintenance and implementing new changes have been experienced to be challenging. Re-engineering is made by replacing old system based on new functional requirements but also reverse engineering system's source code. The re-engineering process involves changing object-oriented Java language to functional and more expressive Clojure.

Software architecture was re-designed and programming language was changed to more reusable one which fits better to data manipulation. With these changes, we achieved a simpler technical structure and a notable reduction in lines of source code. Based on software metrics, survey and interviews, re-engineered software was a success especially in the areas of maintenance and performance. Functional programming language was suitable in web-based system which contains data manipulation.

Result was clearly a better system. The most important goals were better maintainability, ability to correct inaccurate data and better performance. These goals were achieved. Migrating data from the old system to the new one was laborious and required more work than it was estimated. After all, replacement was a success and functional paradigm has its place in implementing web applications and in software that involve programmatical data manipulation. Still, however, it is important to remember to design a good and quality software and use good programming conventions.

Keywords: software re-engineering, functional programming, web application

ALKUSANAT

Diplomityön tekemistä voi luonnehtia projektiksi. Aloittelin sitä ensimmäisen kerran jo vuosia sitten, mutta se oli suurimman osan ajasta tauolla. Aikaa kului ja eteen tuli toinen yrittämä, jonka myötä tekeminen konkretisoitui Netum Oy:ssä työn ohella tehtäväksi liittyen osallistumaani projektiin. Matkan varrella on tapahtunut tutkintouudistus ja viimeisimpänä asiana jopa opinahjoni nimi on muuttunut – niin mukavaa kuin olisikin ollut valmistua teknillisestä yliopistosta.

Haluan kiittää professori Kari Systää työni ohjauksesta ja hyvistä ja rakentavista kommenteista sekä Eero Kuitusta ohjauksesta työpaikalla ja avustuksesta alkuun pääsemiseksi. Lisäksi haluan kiittää tämän diplomityöprojektin tutkimukseen ja haastatteluihin osallistuneita alojensa ammattilaisia ajastaan ja asiantuntemuksestaan sekä lähimpiä kollegoitani herkeämättömästä mielenkiinnosta työni edistymistä kohtaan. Erityiskiitos kuuluu vanhemmilleni, jotka ovat lannistumatta kannustaneet diplomityön tekemiseen ja muistuttaneet sen tärkeydestä diplomi-insinöörin tutkinnon kannalta. Kiitos myös kaikille muille minua auttaneille, mutta erikseen mainitsemattomille henkilöille.

Projektin määritelmään kuuluu aloituksen lisäksi oleellisena osana myös sen lopetus. Tämän diplomityön tekemisen myötä päättyy myös allekirjoittaneen tämänhetkinen opiskelu-ura valmistumisen häämöttäessä aivan nurkan takana. Kiitos kaikille tukena olleille.

Tampereella, 23.10.2019

Jermu Mäki

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

Clojure	Funktionaalinen erillisessä ajoympäristössä ajettava ohjelmointikieli
Java	Suosittu alustariippumaton olio-ohjelmointikieli
JSON	JavaScript Object Notation. Tekstipohjainen datan esitysmalli.
JVM	Java Virtual Machine eli Java-virtuaalikone on ohjelma, joka tulkitsee Java-tavukoodin tietokoneen ymmärtämään muotoon ja toimii samalla ohjelman ajoympäristönä
MVC	Model, view, control. Tekninen arkkitehtuurimalli, jossa ohjelma on jaettu käyttöliittymän sisältävään näkymäkerrokseen, sovelluslogiikan sisältävään kontrollikerrokseen ja tiedot varastoinnin hoitavaan mallikerrokseen.
Paradigma	Ajattelumalli tai suuntaus
PDF	Portable Document Format. Siirrettävä tiedostomuoto esimerkiksi dokumenttien esittämiseen
REPL	Read-eval-print loop. Interaktiivinen komentorivipohjainen ohjelma, jonka avulla voi suorittaa ohjelmakoodia
Sovellus(ohjelma)	Tietokoneohjelma, jonka tehtävänä on ratkaista jokin asia
SPA	Single-Page Application. Web-sivu, joka sisältää yksinkertaisen HTML-sivun lisäksi JavaScript-koodia, jota käytetään datan lataamiseen ja käyttöliittymän päivitykseen.
Web	Yleisnimitys World Wide Webille (WWW), joka yhdistää internetissä olevaa tietoa dokumenttien ja linkkien avulla.

1. JOHDANTO

Ohjelmistoja käytetään yhä enemmän ja niitä hyödynnetään mitä erilaisimmissa asioissa. Niiden käytön tavoitteena on monesti työn tehostaminen, ohjaus ja kustannusten vähentäminen. Ohjelmistojen koot ovat kasvaneet ja niistä on tullut monimutkaisempia. Monimutkaisuutta lisäävät ohjelmistojen sisältämä lisääntynyt toiminnallisuus ja ohjelmistojen väliset keskinäiset integraatiot.

Ohjelmistojen elinkaari kattaa ajanjakson ohjelmiston tekemisen aloittamisesta sen käytön lopettamiseen. Elinkaaren vaiheet voidaan yleistää ohjelmiston määrittelyyn, toteutukseen ja ylläpitoon. Määrittely ja toteutus kuvaavat elinkaaren alkupäätä. Ylläpitoon sen sijaan kuuluu ohjelman käyttöönoton jälkeiset muutokset. Ylläpitovaiheessa muun muassa korjataan ohjelman virheitä ja alun perin väärinmääriteltyjä ominaisuuksia ja vastataan muuttuneisiin tarpeisiin. Elinkaarikustannuksista noin kaksi kolmasosaa kohdistuu ylläpitoon [29].

Ylläpitovaiheessa voidaan lisätä sellaisia ominaisuuksia, joita ei ole alun perin osattu suunnitella tai muutokset voivat olla jopa ristiriidassa alkuperäisten suunnitelmien kanssa. Muutosten tekeminen kasvattaa ohjelmiston toteutuksen monimutkaisuutta ja joustavuus vähenee. Kustannukset, työmäärä ja virheiden todennäköisyys kasvavat. Lopulta ohjelman korvaus on kustannustehokkaampaa kuin muutosten tekeminen. [37]

Ohjelmistojen uudistamisella tarkoitetaan ohjelmiston ylläpitovaiheen laadullista parantamista. Sen sisäisiä tietorakenteita voidaan muuttaa, mutta ohjelmiston toiminnallisuus säilyy pääosin alkuperäisenä. Uudistamisen tavoitteena voi olla myös ei-toiminnalliset ominaisuudet kuten joustavuuden lisääminen. [34] Ohjelmiston korvaus tarkoittaa sen kokonaisvaltaista uudistamista.

Tässä diplomityössä tutkitaan, mitkä seikat vaikuttavat ohjelmistojen uudistamiseen päätymiseen ja mitä uudistamisella tavoitellaan. Lisäksi työssä vastataan kysymyksiin, millä tavalla uudistus tehdään ja miten saavutetaan onnistunut lopputulos. Lopputuloksen arvioinnissa perehdytään laadulliseen ja määrälliseen analyysiin.

Tarkasteltavana tapauksena on laaja web-järjestelmä, jonka erään osakokonaisuuden ylläpidon ja jatkokehittämisen on arvioitu olevan liian haastavaa kustannuksiin nähden. Osakokonaisuus on päätetty uudistaa. Uudistuksen tavoitteena on helpottaa ylläpitoa,

parantaa suorituskykyä, ohjelmiston ymmärrettävyyttä ja jatkokehitettävyyttä, luoda ohjelmistosta joustavampi sekä yksinkertaistaa päivitysprosessia. Tavoitteeseen on tarkoitus päästä korvaamalla osakokonaisuus. Hankalaksi koettuja tietorakenteita yksinkertaistetaan ja sovelluslogiikka eristetään muusta sitä tukevasta toteutuksesta. Ohjelman ymmärrettävyyteen vaikuttavat teknisten rakenteiden monimutkaisuus ja koodirivien määrä. Näiden parantamiseksi uusi sovellus suunnitellaan uudelleen ja toteutetaan funktionaalisella ohjelmointikielellä aikaisemman oliopohjaisen ohjelmointikielen sijaan. Uuden ohjelmointikielen avulla pyritään tuottavampaan tekemiseen keskittymällä tiiviimmin ohjelman sovelluslogiikkaan. Uudistettava osakokonaisuus on liitetty kiinteästi varsinaiseen ydinsovellukseen, jolloin päivityksien tekeminen tuotantoympäristöön edellyttää aina koko järjestelmän päivittämistä. Osakokonaisuus erotetaan ydinsovelluksesta omaksi palvelukseen, jolloin päivitykset voidaan tehdä helpommin ja osan jatkokehittäminen on helpompaa.

Työssä toteutetaan uudistus ja tutkitaan ja arvioidaan, miten hyvin se onnistuu. Tapaustutkimuksen kohteena käytetään Netum Oy:n ohjelmistoprojektia. Onnistumisen arvioinnissa käytetään hyödyksi varsinaisen työn tekemisen aikana tehtyjä havaintoja ja muistiinpanoja sekä ohjelmistokehittäjien ja tuoteomistajan haastatteluja. Arvioinnin tukena käytetään myös suoraan ohjelmakoodista mitattavia asioita kuten koodirivien määrää ja sovelluksen sisäisen rakenteen monimutkaisuutta.

Tämä diplomityö jakaantuu kahdeksaan lukuun. Luvussa kaksi kuvataan ohjelmistojen uudistamista yleisesti sekä uudistamisen tekemistä. Luvussa kolme kerrotaan ohjelmointiparadigmoista, joista keskitytään olio-ohjelmointiin sekä funktionaaliseen ohjelmointiin. Näiden paradigmojen erojen hahmottaminen auttaa ymmärtämään esimerkkitapausta ja siihen liittyvien valintojen tekemistä. Luvussa neljä esitellään web-toimintaympäristö ja web-teknologioita, jotka ovat keskeisessä roolissa uudistettavassa järjestelmässä. Seuraavassa luvussa kuvataan tutkimuksessa käytettävät menetelmät. Luku kuusi kuvaa esimerkkitapauksen ja siihen liittyvän toimintaympäristön. Luku seitsemän kertoo saadut tulokset sekä sisältää tulosten analysoinnin. Luku kahdeksan esittää yhteenvedon ja pohdintaa lopputuloksista.

Aihealueeseen liittyy runsaasti lyhenteitä ja englanninkielisiä termejä, joille ei ole vakiintuneita suomenkielisiä termejä. Kaikki lyhenteet on avattu ensimmäisen kerran mainittuna. Englanninkieliset vastineet on kerrottu termin esittelyn jälkeen suluissa kursiivilla.

2. OHJELMISTON UUDISTAMINEN

Ohjelmistot vanhentuvat väistämättä. Toimintaympäristöt muuttuvat, teknologiat vanhenevat, laitteisto vanhanaikaistuu tai rikkoutuu, millä on vaikutuksia suoritettaviin ohjelmistoihin. Ohjelmistoja pyritään ylläpitämään ja vastaamaan muuttuneisiin vaatimuksiin.

Mitä suurempi ohjelmisto on, sitä kauemmin se on käytössä ja sitä enemmän kuluu resursseja sen ylläpitoon. Tutkimuksen [29] mukaan ylläpitokustannuksista noin 40 % liittyy uusiin vaatimuksiin ja noin 20 % erilaisiin korjauksiin.

Ajan kuluessa uudet muutokset tulevat poikkeamaan niin paljon alkuperäisistä, että niiden muutostyö ei enää ole kannattavaa verrattuna uudelleen tekemiseen. Tässä tapauksessa päädytään ohjelmiston uudistamiseen. Uudistus voi koskea vain tiettyä osaa alkuperäisestä ohjelmistosta, se voi olla ohjelmiston korvaus tai mitä tahansa näiden väliltä. Uudistus voi sisältää toiminnallisia muutoksia, mutta se voi koskea myös ainoastaan ei-toiminnallisia vaatimuksia kuten suorituskykyä. Uudistamisen tavoitteena on tuottaa lisäarvoa uudistettavalle ohjelmistolle. Lisäarvoa voi olla esimerkiksi ohjelmiston laadun parannus tai parempi ymmärrys uudistetusta ohjelmistosta.

Ohjelmistoa voidaan uudistaa monella eri tasolla. Eräs jaottelu on esitetty taulukossa 1. Jaottelu on järjestetty tehtävien muutosten, kustannusten, hyötyjen ja riskien mukaan alkaen pienimmistä. Myös tehtävän työn määrä kasvaa, mitä suurempi uudistuksen taso on.

Taulukko 1 *Ohjelmiston uudistamisen jaottelu [22]*

Uudistuksen taso	Kuvaus
Ylläpito (*continued maintenance*)	Käytössä olevaan ohjelmistoon tehdään normaaleja ylläpitotoimia, mutta siihen ei tehdä suuria muutostöitä
Käyttöliittymäuudistus (*revamp*)	Ohjelmiston käyttöliittymä uudistetaan, jolloin ohjelmisto näyttää uudelta loppukäyttäjille. Ohjelmiston sisäisiin rakenteisiin ei kosketa.
Restrukturointi (*restructure*)	Ohjelman sisäisiä rakenteita muutetaan, mutta ulospäin näkyviä muutoksia ei tehdä.
Arkkitehtuuriuudistus (*rearchitecture*)	Ohjelmisto muunnetaan uuteen teknologiseen arkkitehtuuriin.
Uudelleensuunnittelu hyödyntämällä vanhaa (*redesign with reuse*)	Suunnitellaan ohjelmisto uudelleen, mutta hyödynnetään vanhoja järjestelmäkomponentteja.

Korvaus (*replace*)	Koko järjestelmä korvataan uudella.

2.1 Uudistamiseen johtavia tekijöitä

Ohjelmistoevoluutiotutkimuksen uranuurtaja Meir Lehman on luokitellut ohjelmat S-, P- ja E-tyyppeihin. Luokittelu perustuu ohjelmiston tavasta mallintaa ja abstrahoida ympäristöä. S-tyyppisten ohjelmien määrittelyt on mahdollista laatia etukäteen ja määrittelyt voidaan todistaa oikeaksi. Tällaiset ohjelmat esimerkiksi ratkaisevat matemaattisia kaavoja. P-ohjelmat toteuttavat proseduureja, jotka pyrkivät ennustamaan niin suuria tai monimutkaisia ongelmia, joita ei voi eksaktisti ratkaista. Tyypillinen esimerkki on sään ennustaminen, jossa on niin paljon muuttujia, että ohjelman mallinnus on väistämättä yksinkertaistus todellisuudesta. E-tyyppiset ohjelmat pyrkivät kuvaamaan ihmisten tai yhteiskunnan käyttäytymistä. Näiden ohjelmien ympäristö vaikuttaa vahvasti niiden määrittelyyn. E-tyyppisten ohjelmien pitää jatkuvasti seurata muuttuvia vaatimuksia ollakseen käytettäviä. [37] Tässä työssä keskitytään E-tyyppisiin ohjelmiin ja niiden uudistamiseen.

Lehman on laatinut alun perin ohjelmistoevoluution neljä lainalaisuutta, joihin oikean elämän ohjelmistotuotannossa törmätään [37]. Myöhemmin lait on täydennetty kahdeksaan [36]. Näistä kahdeksasta laista neljä liittyy oleellisesti ohjelmiston uudistamiseen johtaviin syihin. Lait on esitetty listauksessa 1.

Listaus 1 *Ohjelmistoevoluution lakeja*

L1. Jatkuva muutos. Ohjelmistosta tulee vähemmän tyydyttävä, mikäli muuttuneisiin tarpeisiin ei reagoida.

L2. Monimutkaisuuden lisääntyminen. Ohjelmistosta tulee monimutkaisempi, ellei monimutkaisuuden vähentämiseen erikseen panosteta.

L6. Jatkuva kasvu. Ominaisuuksien määrän pitää kasvaa, jotta käyttäjät olisivat tyytyväisiä ohjelmistoon myös jatkossa.

L7. Laadun heikkeneminen. Ohjelmiston laatu heikkenee, jos sitä ei jatkuvasti ylläpidetä ja siihen ei panosteta.

Nämä neljä lakia kuvaavat niitä syitä, jotka johtavat ohjelmiston uudistukseen. Ulkopuolelle jätetyt toiset neljä lakia liittyvät enemmän ohjelmiston tekemiseen ja tekemisen tuottavuuteen, minkä vuoksi ne on jätetty mainitsematta.

Ensimmäinen laki kuvastaa E-tyyppisen ohjelman perusluonnetta, jossa toimintaympäristö toimii takaisinkytkentänä vaatimuksille. Vaatimukset määräytyvät siis käytön perusteella eikä niitä voi täysin tietää etukäteen. Kun vaatimukset muuttuvat, niiden täyttäminen jälkikäteen on vaativaa ja ne saattavat olla ristiriidassa alkuperäisen suunnitelman kanssa. Muun muassa tämän takia ohjelmasta tulee ajan saatossa entistä monimutkaisempi, ellei monimutkaisuutta pyritä aktiivisesti vähentämään. Toinen laki kuvaa tätä monimutkaisuuden lisääntymistä.

Lehmanin kuudes laki on osittain seurausta ensimmäisestä laista. Muutoksien tekeminen johtaa usein uuden ohjelmakoodin lisäämiseen, mikä kasvattaa ominaisuuksien määrää. [33] Toisaalta laki kuvaa myös ihmisten käyttäytymistä. Ohjelmistoon kohdistuvat vaatimukset kasvavat sitä mukaan kuin ohjelmisto on käytössä.

Seitsemäs laki on yhteydessä toiseen lakiin. Toisen lain mukaan monimutkaisuus kasvaa ohjelman elinkaaren aikana. Tämä johtaa laadun heikkenemiseen, ellei ohjelmiston laatua seurata ja siihen kohdenneta kehityskustannuksia ja laatua pyritä aktiivisesti ylläpitämään. [33]

Lehmanin lait perustuvat empiirisiin tutkimuksiin. Samoihin johtopäätöksiin on päädytty myös muissa tutkimuksissa. Eräässä tapaustutkimuksessa [31] ohjelman arkkitehtuurin heikkenemistä oli tapahtunut kehittäjien vaihdosten myötä, koska alkuperäiset suunnitteluperiaatteet eivät ole olleet selviä uusille kehittäjille ja uudesta ohjelmakoodista oli tullut vaikeaselkoista. Koodista oli tullut vaikeampaa muuttaa, sen uudelleenkäytettävyys oli kärsinyt ja sen muokkaaminen oli virhealtista. Toisessa tutkimuksessa [18] alkuperäisen ohjelman korvaamiseen oli päädytty, koska teknologiat olivat niin vanhoja, että ohjelman kytkeminen muihin järjestelmiin oli vaikeaa. Ohjelman jatkokehittäminen oli hankalaa, koska teknologia ei ollut enää yleisesti tuttua ja olemassa olevilla kehittäjillä oli korkea vaihtuvuus.

2.2 Uudistamisen tavoitteet

Uudistamisen tavoitteena on saada hyötyä. Hyötyjä ovat esimerkiksi muuttuneisiin vaatimuksiin vastaaminen, laadunparannus, pienemmät ylläpito- ja käyttökustannukset, suorituskyky, saavutettavuus, modernisointi tai nykyaikaistaminen. Eräs hyöty on myös ohjelman käytöstä aiheutuvien riskien minimointi. Tavoitteena on luoda järjestelmä, joka on kehittyneempi ja jossa on enemmän toimintoja kuin alkuperäisessä järjestelmässä. Uudistuksen tavoitteet voivat olla myös ei-toiminnallisia esimerkiksi ylläpidon parantaminen tai jatkokehityksen helpottaminen.

Maarit Harsu listaa uudistamiselle seuraavia päämääriä: ylläpitokustannusten vähentäminen, ylläpidon helpottaminen, luotettavuuden parantaminen, suorituskyvyn parantaminen ja uudelleenkäytön tehostaminen. Ennen uudistamisprojektin aloittamista pitää arvioida, mitkä sovellukset tai sovelluksen osa-alueet ovat eniten uudistamisen tarpeessa teknisten ominaisuuksien tai liiketaloudellisen arvon perusteella. Päämäärien asettamisen lisäksi pitää määritellä kriteerit ja ehdot, joiden avulla projektin onnistumista voidaan arvioida. Ehtoja kutsutaan onnistumistekijöiksi (*critical success factor*). Kustakin päämäärästä pitää selvittää, minkälaisilla metriikoilla sitä voidaan mitata. Päämääriksi valitaan laadullisten mittareiden lisäksi määrällisiä mittareita. [34]

Taulukossa 1 esitellyillä uudistuksen tasoilla on vaikutusta tavoitteiden asettamiseen. Uudistuksen tason ollessa ylläpitoa, tavoite on säilyttää ohjelman nykyinen laatu, mutta vastata muuttuneisiin vaatimuksiin. Käyttöliittymäuudistuksen tavoite on nimensä mukaisesti uudistaa käyttöliittymä, jolloin tavoitteet ovat esimerkiksi parempi käyttäjäkokemus ja selkeämpi toiminta. Restrukturoinnin tavoitteena on parantaa ohjelman sisäistä ymmärrettävyyttä ja koodin laatua. Arkkitehtuuriuudistuksella on samoja tavoitteita kuin restrukturoinnilla, mutta näiden lisäksi tavoite voi olla parempi jatkokehitettävyys ja uudelleenkäytettävyys. Uudelleensuunnittelulla ja ohjelmiston korvauksella tavoitellaan kokonaisvaltaisesti parempaa ohjelmistoa.

Mainitut tavoitteet ovat vain esimerkkejä ja jokaisessa uudistusprosessissa on omat tavoitteensa, joiden asettaminen ja seuranta on tärkeää. Tavoitteet asettavat ohjenuoran uudistamisprojektille ja niiden seuraaminen on edellytys uudistuksen arvioinnille.

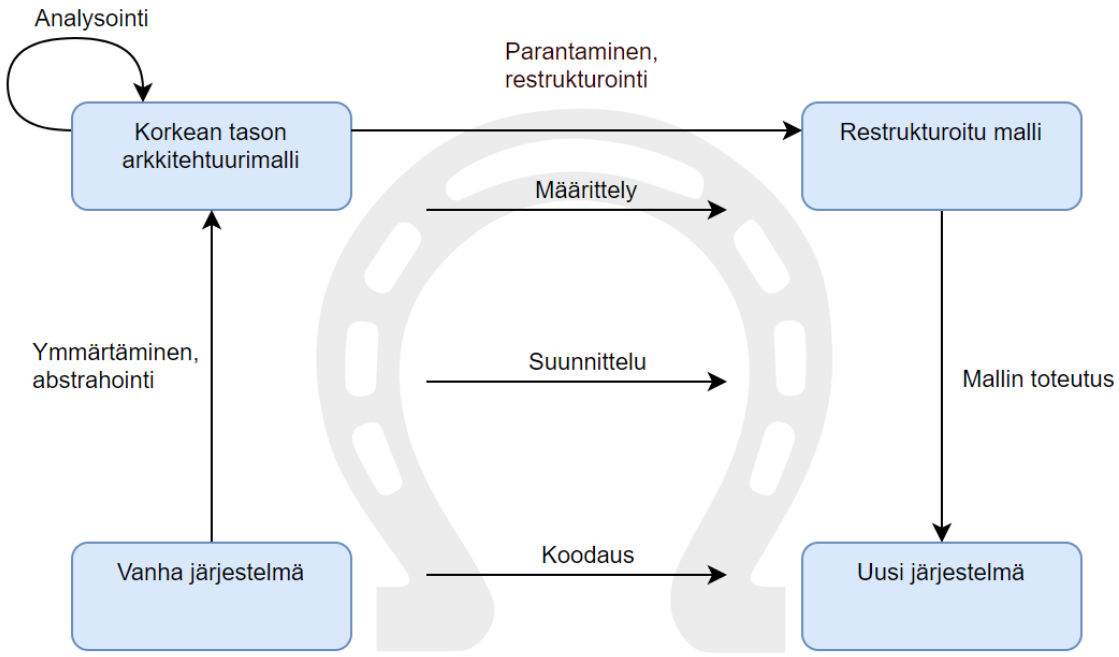
Uudistuksen onnistumisen arvioiminen voidaan tehdä arvioimalla, miten hyvin uudistettu ohjelmisto täyttää sille asetetut tavoitteet ja vastaa uusia vaatimuksia verrattuna alkuperäiseen ohjelmistoon. Arvioiminen ei välttämättä ole helppoa, koska uusi ohjelmisto voi toteuttaa osan vaatimuksista huonommin ja osan paremmin kuin alkuperäinen.

2.3 Uudistamisen tekeminen

Uudistusprosessi jaetaan yleensä kolmeen vaiheeseen. Ensimmäinen vaihe on takaisinmallinnus, jonka tavoitteena on saada ymmärrys ohjelman arkkitehtuurista tai ominaisuuksista. Tämä on tarpeellinen vaihe, mikäli dokumentaatiota ei ole tai se on puutteellista, teknologinen alusta on vanhentunut eikä sitä voi käsitellä työkaluilla tai jos alkuperäisiä kehittäjiä ei ole saatavilla. Takaisinmallinnuksessa selvitetään ja pyritään ymmärtämään, miten ohjelma toimii, mutta siihen ei sisälly muutosten tekemistä. Tätä vaihetta seuraa ohjelmiston rekonstruointi, jossa toimitaan arkkitehtuurisuunnitelman tasolla. Tar-

koituksena on parantaa järjestelmän arkkitehtuurisuunnitelmaa ja tietomalleja, joista tulee muodostumaan haluttu uusi arkkitehtuuri. Arkkitehtuurisuunnitelma evaluoidaan huomioiden nykyiset ja tulevat järjestelmän laatuvaatimukset. Viimeisenä vaiheena on rekonstruoidun suunnitelman toteutus. [55]

Uudistamisen prosessimalli noudattaa niin sanottua hevosenkenkämallia [45]. Nimitys tulee etäisesti hevosenkenkää muistuttavasta visualisoinnista. Prosessin yksinkertaistettu malli on esitetty kuvassa 1. Uudistusprosessi alkaa vasemmasta alakulmasta ja jatkuu myötäpäivään yläkautta oikeaan alakulmaan. Siniset laatikot kuvaavat vaiheista syntyviä lopputuloksia. Abstraktiotaso on sitä korkeampi, mitä ylemmällä tasolla toimitaan.



Kuva 1 *Uudistamisen prosessimalli [45], [55]*

Uudistamiseen voi käyttää työkaluja. Työkalut soveltuvat esimerkiksi takaisinmallinnukseen olemassa olevasta lähdekoodista. Takaisinmallinuksen lopputuloksena syntyy korkeamman abstraktiotason kuvaus, jonka avulla selviävät moduulien väliset keskinäiset suhteet. Työkalujen käyttö korostuu, mikäli dokumentaatio on vain vähäistä.

2.4 Riskit

Ohjelmiston uudistamiseen kuten kaikkiin ohjelmistoprojekteihin liittyy riskejä. Vanha järjestelmä on olemassa ja se on vähintäänkin jossain määrin käytettävä. Uudistus on investointi parempaan. Uudistusprojektiin liittyy riskien kartoitus, niiden vaikutusten arviointi, ennaltaehkäisy sekä seuranta.

Riskeihin vaikuttavat useat asiat. Uudistettavat ohjelmistot ovat usein vanhoja, jolloin useat eri henkilöt voivat olla olleet osallisina ohjelman kehittämiseen. Henkilöiden vaihtuessa tieto alkuperäisistä suunnitteluperiaatteista voi heikentyä. Tiedonvälitys korostuu entisestään, jos ohjelman dokumentointi on heikkoa tai sitä ei ole pidetty ajan tasalla. Ohjelma voi olla kasvanut ajan myötä niin suureksi, että sen tekninen ymmärtäminen voi olla hankalaa. [12] Uudet kehittäjät voivat tehdä muutoksia, jotka ovat ristiriitaisia alkuperäisiin periaatteisiin nähden, mikäli he eivät tiedä suunnitteluperiaatteiden alkuperäisiä tarkoituksia [31].

Riskit jakaantuvat eri osa-alueisiin. Uudistamisprosessi on ensimmäinen osa-alue. Riskinä on, että prosessia ei suunnitella kunnolla tai riittävän huolellisesti eikä sille varata riittävästi aikaa. Uudistamista tehdään muun toiminnan ohessa eikä kukaan ole varsinaisesti vastuussa. Tällöin uudistamiseen ei sitouduta. Uudistamisprosessi voi tässä tapauksessa kestää liian kauan, jolloin uusi järjestelmä voi olla valmistuessaan vanhanaikainen. Myös uudistamisen tavoitteet voivat jäädä saavuttamatta. Vaikka ohjelma toimisi muuten, niin loppuvaiheessa voidaan havaita, että esimerkiksi suorituskyky voi olla riittämätön. [34]

Toinen osa-alue liittyy takaisinmallinnukseen. Ohjelman alkuperäisten vaatimusten tunnistaminen voi olla odottamattoman vaikeaa. Koodiin sulautettua liiketoiminnallista tietoa on vaikea löytää. [34] Määrittely- ja suunnitteludokumentaatio ei ole välttämättä ajan tasalla tai se on puutteellista. Ohjelman lähdekoodi pitää sisällään kattavimman kuvauksen toiminnallisuudesta ja erilaisista ohjelman sisäisitä säännöistä, mutta sen tutkiminen edellyttää vahvaa ymmärrystä ja on työlästä.

Kolmas osa-alue liittyy uuden järjestelmän toteuttamiseen. Toteuttamisen yhteydessä yritetään lisätä liikaa uusia vaatimuksia tai toimintoja. Tietorakenteiden siirtäminen uuteen ympäristöön saattaa olla vaikeaa. Vanhasta järjestelmästä löydetyt tekniset ratkaisut tai piirteet eivät välttämättä sovellu uuteen järjestelmään. Myös uudistamisessa käytettäviin ohjelmistotyökaluihin liittyy riskejä. Ne eivät välttämättä toimi odotetulla tavalla tai niitä ei ylipäätään ole saatavilla. [34] Uudistetussa sovelluksessa voi olla käytön estäviä virheitä tai pienempien virheiden lukumäärä on liian suuri. Toisaalta täysin virhee-

tön ohjelma voi toimia väärin eikä loppukäyttäjien haluamalla tavalla. Uudelleensuunniteltu arkkitehtuuri voi osoittautua sopimattomaksi uudistusprojektin aikana. Pahimmassa tapauksessa arkkitehtuurilliset heikkoudet ilmenevät vasta tuotantokäytössä. [12]

Viimeinen osa-alue liittyy toimintatapoihin. Järjestelmän uudistamiseen on sitouduttu ennenaikaisesti, ennen kuin on selvitetty tarvittavan ohjelman uudistamisen laajuus. Pitkän tähtäimen tavoitteet voivat puuttua tai välitavoitteita ei ole. Asetetut päämäärät voivat olla epärealistisia. Uudistamistapa tai -laajuus ei välttämättä sovi yhteen yrityksen päämäärien, budjetin tai aikataulun kanssa. [34]

Mainitut riskit liittyvät erityisesti ohjelmistojen uudistamiseen. Niiden lisäksi uudistamiseen liittyvät myös kaikki samat riskit kuin mihin tahansa ohjelmistoprojektiin kuten henkilöstöön liittyvät riskit osaamisen tai vaihtuvuuden osalta.

3. OHJELMOINTIPARADIGMOJA

Ohjelmointikielten historia ulottuu 1940-luvun loppuun, jolloin nykyisenkaltaisten tietokoneiden esiinmarssi alkoi [32]. Aluksi ohjelmoitiin konekielellä, joka koostui binääriluvuista, siis ykkösistä ja nollista, ja oli suoritintyyppiriippuvaista. Konekieli kehittyi 1960- ja 1970-lukujen vaihteessa assembly-kieleksi, joka mahdollisti symbolien kirjoittamisen lukujonojen sijaan. Tämä mahdollisti myös ohjelman järjestämisen rakenteelliseksi ja proseduraaliseksi. Lopulta proseduraalinen ohjelmointikieli on väistynyt olio-ohjelmoinnin tieltä. [14] 1990-luvulla internetin yleistyttyä skriptikielet tulivat yleisiksi [35].

Nykyisin ohjelmointikieliä on useita kymmeniä, ellei satoja. Mikään kielistä tai paradigmoista ei ole yksiselitteisesti toistaan parempi vaan soveltuvuus on aina tapauskohtaista. Tässä luvussa tarkastellaan tarkemmin kahta ohjelmointiparadigmaa; olio-ohjelmointia ja funktionaalista ohjelmointia. Näiden paradigmojen ymmärtäminen auttaa taustoittamaan työn lähtökohtaa ja tavoitteiden ymmärtämistä.

3.1 Olio-ohjelmointi

Olio-ohjelmointi on tällä hetkellä käytetyimpiä ohjelmointiparadigmoja. Olio-ohjelmointiparadigmaa noudattavia ohjelmointikieliä ovat muun muassa Java, C++, Python, C# ja JavaScript.

Olio-ohjelmoinnissa keskeisenä käsitteenä ovat nimensä mukaisesti oliot (*object*). Oliot yhdistävät datan ja metodit. Metodit kuvaavat olion käyttäytymistä ja niiden avulla suoritetaan esimerkiksi datan manipulointia ja järjestelmän sovelluslogiikkaa. Luokat (*class*) kuvaavat olioita ja ne voikin mieltää olioiden tyypeiksi. Ohjelmakoodissa kuvataan luokat, joista ohjelman suoritusaikana luodaan käytettävät oliot. Oliot ovat luokkien ilmentymiä. Ohjelman sovelluslogiikka perustuu olioiden toimintaan toisin kuin proseduraalisissa kielissä, joissa keskiössä ovat funktiot ja logiikka.

Esimerkki luokan määrittelystä on kuvattu ohjelmassa 1. Luokka kuvaa henkilöä. Jokaisella henkilöllä on nimi ja syntymäaika. Tällä luokalla voi olla useita ilmentymiä eli olioita, joilla jokaisella on oma nimi ja syntymäaika. Sovelluslogiikka näiden ominaisuuksien ympärillä on kuitenkin sama riippumatta niiden arvoista. Luokassa on metodi, jonka avulla on mahdollista laskea jokaisen henkilön yksilöllinen ikä.

```
public class Person {
   private long birthYear;
   private String name;

   public Person() {}

   public void setBirthYear(long year) { this.birthYear = year; }

   public long getAge() {
      return LocalDate.now().getYear() - this.birthYear;
   }
}
```

Ohjelma 1 *Esimerkki luokasta*

Olio-ohjelmointi on kehittynyt periaatteesta, jossa vähennetään monimutkaisuutta abstraktion avulla. Proseduraalisissa kielissä abstraktiota voi lisätä proseduureilla ja funktioilla. Olio-ohjelmoinnissa abstraktiotasoina ovat luokat ja oliot. [14]

Kolme olio-ohjelmoinnin perusideaa ovat kapselointi, periytyvyys ja polymorfismi [14]. Kapselointi tarkoittaa tässä tapauksessa monimutkaisuuden piilottamista olioiden taakse. Olioiden käyttäjän ei tarvitse tietää niiden sisäisestä toteutuksesta. Periytyvyyden avulla luokat voivat muodostaa hierarkioita, joissa alemmalla tasolla olevat luokat perivät ominaisuuksia ylemmältä tasolta. Polymorfismilla tarkoitetaan, että samankaltaisten olioiden julkinen rajapinta voi olla sama, mutta rajapinnan toteutus on erilainen.

3.1.1 Olio-ohjelmoinnin hyötyjä

Olio ohjelmoinnilla on selkeitä hyötyjä. Hyödyt voidaan tiivistää kolmeen eri osa-alueeseen; koodin ylläpidettävyyteen, uudelleenkäytettävyyteen ja laajennettavuuteen. [10]

Yksinkertaisen ja pienen ohjelman voi kirjoittaa millä kielellä tahansa, mutta mitä suuremmaksi ohjelma kasvaa, sitä enemmän hyvä ohjelmointityyli korostuu. Huonoilla ohjelmointityylin tekniikoilla ohjelmassa ilmenee virheitä sitä myötä kuin vanhoja saadaan korjattua. Eräs syy, joka aiheuttaa ennalta-arvaamattomia virheitä, johtuu ohjelman sisäisten osien välisistä riippuvuuksista. Olio-ohjelmointi tarjoaa sisäänrakennetun keinon tehdä ohjelmista modulaarisia rinnastamalla luokat moduuleiksi. [10] Modulaarisuus helpottaa ylläpitoa, koska ohjelma jakaantuu erillisiin ja itsenäisiin osiin, joiden ymmärtäminen on helpompaa. Pienempien kokonaisuuksien uusiminen on usein helpompaa, koska ymmärrettävää on vähemmän ja muutokset ovat pienempiä.

Ohjelmistot ovat tulleet yhä suuremmiksi, monimutkaisemmiksi ja vaativammiksi. Ohjelmiston uudelleenkäytöstä on tullut merkittävä kustannuksia alentava tekijä. [29] Uudelleenkäytettävyys tarkoittaa, että aikaisemmin kirjoitettua ohjelmakoodia voidaan hyödyntää uudessa samanlaisen toiminnallisuuden sisältävässä osassa. Kaikkea ei tarvitse tehdä uudestaan. Uudelleenkäytettävyyteen liittyy myös ohjelmakoodin lisäksi uudelleen toistuvat suunnitteluperiaatteet. [10]

Joskus voi tulla vastaan tilanteita, joissa olemassa olevaa toiminnallisuutta ei voi suoraan käyttää uudelleen. Tällöin toiminnallisuutta voidaan laajentaa. Olio-ohjelmoinnissa laajennettavuus saavutetaan perimisellä (*inheritance*), jossa ylempien luokkien ominaisuudet ja toiminnallisuus välittyvät aliluokkiin. Toiminnallisuuden perivissä luokissa voidaan lisätä toiminnallisuutta tai muuttaa olemassa olevaa. [10]

Erään ajatuksen mukaan olio-ohjelmointi mahdollistaa ohjelmistoarkkitehtuurin, jossa korkean tason moduulien linjaukset ja toimintatavat on erotettu matalamman tason moduulien yksityiskohdista. Matalamman tason osia voidaan kehittää erikseen ja itsenäisesti. [46]

3.1.2 Java

Java perustuu olio-ohjelmointiparadigmaan. Se on Sun Microsystemsin kehittämä ja alun perin kehitetty 1990-luvulla. Tavoitteena oli kehittää ohjelmointikieli, joka olisi uusi C++-toteutuksen kaltainen kieli. Java-ohjelmat voitaisiin suorittaa laitteissa, jotka olisivat yhdistettynä verkkoon ja joille käyttöliittymänä toimisi televisio. Java-ohjelmat lähetettäisiin palvelimilta palvelun käyttäjien asiakasohjelmiin, joissa ohjelma suoritettaisiin. Tällä tavoin palvelimien suoritusresursseja voitaisiin hajauttaa. Johtavia suunnitteluperiaatteita olivat siirrettävyys (*portability*) ja tietoturvallisuus (*security*). [32]

Siirrettävyys oli tärkeää, koska ohjelmia voitiin suorittaa missä tahansa ympäristössä. Suoritusympäristön moninaisuuden ratkaisuksi esitettiin Java-virtuaalikone (Java Virtual Machine, JVM) ja sillä suoritettava tavukoodi. Java-ohjelma käännetään ensin tavukoodiksi, jonka virtuaalikone tulkitsee suorituksen aikana. [32] Virtuaalikone on alustariippuvainen toisin kuin Java-ohjelma.

Tietoturvallisuuteen liittyviä tekijöitä ovat muun muassa luotettavuus ja toimintavarmuus sekä ohjelmointikielen yksinkertaisuus, joka osaltaan vähentää virheiden määrää. Java on vahvasti tyypitetty kieli, jossa tyyppitarkastuksia tehdään kolmella eri tasolla; käännösaikaisesti, tavukoodin tasolla sekä ajonaikaisesti. Javassa ei ole manuaalista osoittimien (*pointer*) hallintaa vaan muistinhallinta hoidetaan virtuaalikoneen roskienkeruun (*carbage collection*) avulla. [32]

Javan ja olio-ohjelmoinnin avulla voidaan tehdä kapselointia. Esimerkki tästä on kuvattu ohjelmassa 2. Ensimmäisellä rivillä luodaan henkilö-olio, joka on henkilö-luokan ajonaikainen ilmentymä. Henkilö-oliolle voi asettaa syntymävuoden, jonka jälkeen tämänhetkinen ikä on kysyttävissä. Olion käyttäjän ei tarvitse tietää, millä tavalla iän laskeminen tapahtuu. Luokka tarjoaa julkisen rajapinnan. Julkisen rajapinnan lisäksi luokka voi sisältää sisäisiä metodeja, joita ei voi käyttää kuin luokan sisältä. Kapseloinnin avulla voidaan piilottaa tarpeettomat yksityiskohdat ulkopuolisilta osilta.

```
Person person = new Person();
person.setBirthYear(1995);
person.getAge();  // => 24
```

Ohjelma 2 *Esimerkki kapseloinnista ja yksityiskohtien piilottamisesta*

Java on tällä hetkellä maailmanlaajuisesti suosituimpia kieliä. Avoimen lähdekoodin julkaisualustan GitHubin selvityksessä [41] Java on toiseksi suosituin kieli, ohjelmointiyhteisö Stack Overflown kyselyssä [15] Java on viidenneksi suosituin ja ohjelmiston laatuun keskittyvän yrityksen TIOBE:n tutkimuksessa [56] Java on suosituin.

3.2 Funktionaalinen ohjelmointi

Funktionaalisen ohjelmoinnin historia ulottuu 1950-luvun loppupuolelle, jolloin John McCarty julkaisi ensimmäisen funktionaalisen ohjelmointikielen LISP:n. LISP pohjautuu kuitenkin jo 30 vuotta tätä aikaisempaan Alonzo Churchin ja kollegoiden kehittämään lambda-kalkyyliin, joka kuvaa laskentaa funktioiden ja niiden suorituksen avulla. [26] Myöhemmin julkaistuja funktionaalisia kieliä ovat muun muassa Scheme, ML (Meta Language) ja Haskell. Näistä viimeisintä voidaan pitää puhtaana funktionaalisena kielenä. [35]

Funktionaaliseen ohjelmointiparadigmaan kuuluu datan muuttumattomuus (*immutability*). Tämä lähtökohta on päinvastainen oliopohjaisiin kieliin verrattuna, joissa oliot sisältävät muuttuvaa dataa. Muuttumattomuudella saavutetaan hyötyjä kuten säieturvallisuus. Koska mikään säie ei voi muuttaa dataa, ne eivät voi vaikuttaa toisiinsa muuttuvan datan välityksellä. [3] Jokainen datan käsittelystä johtuva kilpailutilanne (*race condition*), lukkiutuminen (*deadlock*) ja yhtäaikainen päivitys (*concurrent update*) voidaan välttää muuttumattomilla tietorakenteilla. Toisin sanoen, kaikki rinnakkaisuudesta johtuvat ongelmat voidaan välttää muuttumattomuudella. [46]

Toisena funktionaalisen paradigman piirteenä on ensimmäisen luokan funktiot (*first-class functions*). Tällä tarkoitetaan sitä, että funktiot toimivat hyvin samankaltaisesti kuin mikä tahansa muu tietotyyppi. Funktiota voi luoda ajonaikaisesti, välittää eteenpäin parametreina tai muuttujina ja palauttaa toisista funktioista. [3]

Kolmas funktionaalisen ohjelmointiparadigman piirre on, että funktiot ovat puhtaita (*pure*). Niillä ei ole sivuvaikutuksia, jotka voivat muuttaa funktion toimintaa ajonaikaisesti. Puhtaan funktion paluuarvo on aina sama kutsuttaessa sitä samoilla parametreilla. [3] Puhtaita funktioita on erityisen helppoa testata yksikkötestien avulla, koska paluuarvo riippuu vain annetuista argumenteista.

Sovelluskehittäjien kannalta tarkasteltuna, funktionaalisella mallilla ja olio-ohjelmoinnilla on hyvin paljon eroa. Olio-ohjelmointi perustuu funktioiden ja muuttuvan datan yhdistämiseen. Funktionaalisissa kielissä taas data on lähtökohtaisesti muuttumatonta. Kielet voi mieltää toistensa vastakohdiksi.

3.2.1 Clojure

Clojure on dynaamisesti tyypitetty ja funktionaalinen ohjelmointikieli, jossa on pyritty huomioimaan suorituskyky. [3] Clojure on funktionaalisuuden lisäksi symbioottinen ja homoikoninen. Symbioottisella tarkoitetaan, että Clojure on tarkoitettu suoritettavan jonkin toisen osapuolen tarjoamassa ympäristössä. Tyypillisesti Clojurea ajetaan JVM-ympäristössä, mutta myös muita ajoympäristöjä voidaan hyödyntää kuten JavaScript-ajoympäristöä internetselaimessa ja Microsoftin .NET -teknologian Common Language Runtime -ajoympäristöä. Homoikoninisuudella tarkoitetaan, että koodi on samalla dataa, mikä puolestaan helpottaa monipuolisten makrojen tekemistä. [38] Makrot ovat ohjelmakoodia, jotka puretaan esikäännösvaiheessa ja niistä muodostuu uutta ohjelmakoodia.

Clojuren on kehittänyt Rick Hickey. Nimi Clojure on yhdistelmä ohjelmointikielistä C#, Lisp, Java – (C)(L)o(J)ure [38]. Osaltaan nimi on sanaleikki, jossa viitataan kielen syntaksissa esiintyviin sulkeisiin ja ohjelmalohkoihin.

Clojure ei ole puhtaasti funktionaalinen. Clojuren avulla on mahdollista muuttaa ohjelman tilaa käyttämällä erityisiä viittauksia, atomeja. Atomien käsittelyssä on huomioitu monisäikeisyys. Viittauksien käyttäminen on tehty kielen toteutuksen tasolla hallitusti, jolloin funktionaalinen puoli ei juurikaan kärsi.

Clojure-ohjelmasta on mahdollista kutsua Java-koodia ja siinä onkin mahdollista hyödyntää Javalle tehtyjä kirjastoja. Tästä on suurta hyötyä, koska kaikkia Javalle tehtyjä kirjastoja voi suoraan hyödyntää. Haasteena on käyttää olioparadigman toteutusta funktionaalisen paradigman kielen kautta. Lopputulos on kahden keskenään ristiriidassa olevan

paradigman sekoitus. Koodin luettavuus kärsii, mutta hyödyt ovat silti selkeästi suuremmat kuin haitat.

Clojuren syntaksi on saanut vaikutteita Lisp-ohjelmointikielestä. Nimi Lisp tulee sanoista LISt Processing, joka tarkoittaa listojen suorittamista. Funktiokutsut ovat listoja, joissa ensimmäisenä on suoritettava funktio. Tätä seuraa funktiolle annettavat argumentit. Yksinkertainen esimerkki Clojure-syntaksista on neljän luvun yhteenlasku, joka on esitetty ohjelmassa 3. Yhteenlaskufunktion argumenttien määrä ei ole rajoitettu.

```
(+ 1 2 3 4)  ;; => 10
```

Ohjelma 3 *Neljän luvun yhteenlasku Clojurella*

Clojure on ilmaisuvoimainen kieli. Ilmaisuvoimalla tarkoitetaan tässä tapauksessa kykyä keskittyä ohjelmakoodin tasolla sovelluslogiikkaan kielen rakenteiden sijaan. Tiiviydellä tarkoitetaan koodirivien ja koodin määrää. Koodin määrän pienentämisellä on selkeitä hyötyjä. Muutoksien tekeminen on mahdollista pienemmällä työmäärällä ja uuden koodin kirjoittaminen on nopeampaa. Esimerkki Clojuren ilmaisuvoimasta ja tiiviydestä on esitetty ohjelmissa 4, 5 ja 6. Kaikissa lasketaan parametrina annettujen lukujen keskiarvo, mutta koodin määrä on oleellisesti kieliriippuvaista. Clojure- ja Java 7 -koodi eroavat toisistaan kontrollirakenteiden osalta. Java 7:ssa ei ole mahdollista tehdä samanlaista toteutusta eli Clojure on ilmaisuvoimaisempi. Java 8:n koodilla ja Clojure-koodilla ei ole rivimäärän mukaan mitattuna juurikaan eroa. Eroa kuitenkin tulee, mikäli tarkastelu tehdään koodimäärän perusteella. Clojure-koodin määrä on karkeasti laskettuna puolet Java 8:n koodimäärästä. Clojure on siis tiiviimpää. Erot johtuvat pääasiassa Javan pakollisista muuttujien tyyppimäärittelyistä, jotka Clojuressa määräytyvät dynaamisesti.

```
(defn calculate-avg [values]
   (/ (reduce + values) (count values))
```

Ohjelma 4 *Keskiarvon laskeminen Clojurella*

```
static double calculateAvg(int[] values) {
   int s = 0;
   for (int i : values) {
      s += i;
   }
   return s / (double)values.lenght;
}
```

Ohjelma 5 *Keskiarvon laskeminen Java 7:lla*

```
static double calculateAvg(List<Integer> values) {
   return values.stream().reduce(0, (acc, i) -> acc + i)
          / (double)values.size();
}
```

Ohjelma 6 *Keskiarvon laskeminen Java 8:lla*

3.2.2 Java 8:n funktionaaliset piirteet

Javaan on tuotu funktionaalisia piirteitä alkaen versiosta 8. Eräs näkyvimmistä funktionaalisista piirteistä on lambda-lauseke, jota voidaan hyödyntää esimerkiksi tietovirtojen eli streamien kanssa [27]. Ohjelmassa 6 on käytetty näitä molempia. Streamin avulla luodaan laiska tietovirta listan arvoista. Käytettäessä streameja alkuperäiset arvot eivät muutu, joten sen avulla suoritus voidaan jakaa rinnakkain ajettaviin lohkoihin, mikäli sivuvaikutuksia ei ole. Ohjelmakoodin kohdassa `(acc, i) -> acc + i` luodaan anonyymi lambda-lauseke. Tämä lauseke muistuttaa funktiota, mutta sillä ei ole formaalia funktion määrittelyä. Lauseke ottaa parametreikseen kaksi kokonaislukumuuttujaa, joiden tyypit Java-kääntäjä osaa päätellä tässä tapauksessa ilman eksplisiittistä tyyppimäärittelyä. Nämä muuttujat ovat kaarisulkujen sisällä. Lausekkeen toteutus tulee nuolen jälkeen. Toteutuksessa lasketaan annetut parametrit yhteen ja palautetaan tämän laskun tulos. Javan lambda-lausekkeita voi sijoittaa muuttujiin ja antaa parametreina toisin kuin perinteisiä Javan funktioita. Niiden näkyvyysalue määräytyy normaalien muuttujien näkyvyyssääntöjen mukaisesti. Lambda-lausekkeen näkyvyys voi olla toisen funktion sisällä toisin kuin perinteisillä luokkiin ja olioihin sidotuilla funktioilla. Luettavuus paranee ja ohjelmakoodista tulee selkeämpää, koska formaaleja funktion määrittelyjä ei tarvitse tehdä eikä tyyppimerkintöjä tarvita. Tarkemmalla näkyvyysaluerajoituksella voidaan vähentää monimutkaisuutta, mikä osaltaan parantaa ohjelman ylläpidettävyyttä.

3.2.3 Funktionaalinen reaktiivinen ohjelmointi

Reaktiivisella ohjelmoinnilla tarkoitetaan, että ohjelma pohjautuu datavirtoihin, joihin reagoidaan. Datavirtojen sisältö pohjautuu tapahtumiin (*event*), jotka voivat syntyä käyttäjän toimesta tai ne voivat pohjautua esimerkiksi antureihin. Oleellista kuitenkin on, että tapahtumien ilmaantumista ei voi etukäteen tietää varmasti. [48] Ohjelma ikään kuin odottaa tapahtumia, joihin se voisi reagoida. Tapahtumien lisäksi oleellisena käsitteenä on käytös (*behaviour*). Ohjelmakomponentille kuvataan, millä tavalla se käyttäytyy. Käytös voi perustua dataan, jonka muutoksista aiheutuu tapahtumia. Web-aiheisena esimerkkinä voisi olla syöttökentän validointi, joka tehdään kentän arvon muuttuessa. Muutoksesta aiheutuu tapahtuma, joka vaikuttaa komponentin käytöksen kautta ohjelman toimintaan. Käytös perustuu syöttökentän arvoon. Mikäli arvo on epäkelpoinen, näytetään käyttöliittymässä virheellisen arvon ilmaisin.

Funktionaalinen reaktiivinen ohjelmointi (*Functional Reactive Programming,* FRP) lisää reaktiiviseen ohjelmointiin funktionaalisen tyylin. Funktionaalisuudesta lainataan muuttumattomuus ja kyky muodostaa ohjelmakomponenteista sellaisia koosteita, joilla ei ole yllättäviä sivuvaikutuksia. [48] Funktionaalinen reaktiivinen ohjelmointi on ajatus- tai arkkitehtuurimalli, joka voidaan toteuttaa useilla eri ohjelmointikielillä.

4. WEB-OHJELMOINTI

Nykyään yhä suurempi osa sovelluksista toimii joko suoraan webissä tai webin kautta. Tämän on mahdollistanut tietoliikenneyhteyksien lisääntyminen ja parantuminen sekä päätelaitteiden runsas kasvu. Tietokoneiden lisäksi suurella osalla ihmisistä on käytössään älypuhelin, jonka avulla sovelluksia on mahdollista käyttää mistä tahansa ja milloin tahansa. Web-ympäristön tunteminen edellyttää esimerkkitapauksen toimintaympäristön tuntemisessa.

4.1 Web toimintaympäristönä

Web on tarjonnut alun perin tavan päästä käsiksi dokumenttikokoelmiin. Siitä se on kehittynyt infrastruktuuriin, jonka avulla voidaan tehdä monimutkaisia ohjelmistosovelluksia, jotka ovat saatavilla kaikkialla ja joita voidaan käyttää monista eri päätelaitteista. [49] Nykyisin monien asiointipalveluiden käyttöliittymä on tehty webin päällä toimivaksi.

Web teknologiana ja web-sovellukset tarjoavat selkeitä hyötyjä. Eräs oleellinen hyöty on sovelluksen tavoitettavuus. Kuka tahansa voi päästä sovellukseen mistä tahansa. Web-sovellukset eivät ole aikaan, paikkaan tai käytettävään päätelaitteeseen sidottuja.

Tavoitettavuudella on myös varjopuolensa. Ilkeämielisillä käyttäjillä on helppo pääsy järjestelmiin, jotka vaikuttavat useisiin loppukäyttäjiin. Niinpä tietoturvallisuus on korostuneessa roolissa web-toimintaympäristössä. Asian ympärille on muodostunut kokonaisia yhteisöjä kuten kansainvälinen verkkoyhteisö OWASP, joka pitää ajantasaista listaa yleisistä web-haavoittuvuuksista [42].

Web helpottaa sovellusten ylläpitoa. Ohjelmistopäivitys pitää tehdä vain kerran, jonka jälkeen kaikki käyttäjät saavat uuden version käyttöönsä ilman erillisiä toimenpiteitä. Tämä yksinkertaistaa päivitysten jakelumallia. Perinteisen tietokoneohjelman päivittäminen edellyttää jokaisen käyttäjän asentavan päivityksen erikseen. Tätä on vielä edeltänyt päivityksen hankkiminen tavalla tai toisella.

Erään näkemyksen mukaan web ei ole tuonut ohjelmistojen kehittämiseen mitään uutta. Se toimii vain käyttöliittymänä, joka on ohjelmiston kannalta vain yksityiskohta. [46] Kaikesta huolimatta webin käyttö on kasvanut räjähdysmäisesti sen alkuvuosista ja sen päälle tehdään yhä enemmän sovellusohjelmia. Hyvänä esimerkkinä toimivat lukuisat web-sovellukset erilaisista asiointipalveluista sosiaalisen mediaan.

4.2 Asiakas-palvelin-arkkitehtuuri

Asiakas-palvelin-arkkitehtuurissa on kaksi roolia; palveluiden tarjoajat ja niiden käyttäjät. Tämän arkkitehtuurimallin ajatus perustuu jaettuun resurssiin, jonka palveluita tarjotaan käyttäjille. Keskitetyllä palvelulla käyttöä voidaan valvoa suoraviivaisesti. Käyttöön liittyvien teknisten haasteiden hallinta, kuten esimerkiksi samanaikainen muokkaus, on keskitetty palvelun tarjoajalle, ja asiakkaat voivat toimia toisistaan riippumatta. [30]

Web-ympäristössä käyttäjät käyttävät asiakasohjelmien avulla palvelimen tarjoamia palveluja tai resursseja. Tyypillisesti web-ympäristössä asiakasohjelmana käytetään internetselainta, mutta myös muita ohjelmia on käytössä kuten esimerkiksi asiakasohjelmat sähköpostien lukemiseen. Myös monissa mobiilisovelluksissa käyttöliittymänä on natiivi sovellus, mutta tietojen hakeminen ja tallennus tapahtuvat webin avulla.

Web-ympäristössä käyttäjän toimet voidaan ajatella tapahtumavirtana, johon ensisijaisena reagoijana on selaimessa ajettava ohjelma. Palvelinta voidaan ajatella aivan vastaavanlaisena ympäristönä. Se reagoi ulkoisen rajapinnan, tässä tapauksessa esimerkiksi HTTP-rajapinnan, tapahtumiin.

4.3 Web-teknologioita

WWW:n (World Wide Web) tai webin historian aikana sen päälle on rakennettu lähes 30 vuoden aikana satoja, ellei tuhansia eri teknologioita. Osa teknologioista on niin sanottuja matalan tason teknologioita kuten protokollamäärittelyjä tai -standardeja, osa taas kuvaa kokonaisia sovelluskehyksiä. Tässä aliluvussa esitetään uudistettavan ohjelmiston kannalta oleellisimmat teknologiat.

4.3.1 HTTP-protokolla

HTTP-protokolla (HyperText Transfer Protocol) on Tim Berners-Leen ja Robert Cailaun kehittämä standardoitu web-protokolla, jonka avulla asiakasohjelmat ja palvelimet kommunikoivat keskenään. Se perustuu pyyntöihin (*request*) ja vastauksiin (*response*). Tyypillisessä tapauksessa käyttäjä pyytää haluttua resurssia selaimen kautta web-palvelimelta, johon palvelin antaa vastauksen. [50] HTTP on tilaton protokolla [19]. Tilattomuudella tarkoitetaan, että palvelin ei muista aiempia pyyntöjä, jolloin kaikissa pyynnöissä pitää olla kaikki sen käsittelemiseen tarvittava tieto.

4.3.2 HTML ja CSS

HTML (Hypertext Markup Language) on rakenteinen dokumenttien kuvauskieli, jota käytetään web-ympäristössä. HTML kuvaa datan ja sen rakenteen. Se on sisällöltään staattista webin alkuperäisen ajatuksen ja käyttötarkoituksen takia. HTML-merkkaus tulkitaan ja esitetään päätelaitteen avulla, tyypillisesti internetselaimella.

CSS (Cascading Style Sheets) on web-standardi, joka kuvaa esitystavan HTML- tai XML-dokumentille (XML, eXtensible Markup Language). Sen avulla voidaan muuttaa tiedon muotoilua kuten kirjasinta, värejä ja eri osioiden tai merkkien välejä. Uudemmissa versioissa on tullut tuki animaatioille. [11] Vastaavasti kuin HTML, myös CSS tulkitaan internetselaimen avulla.

4.3.3 JavaScript

JavaScript on tulkittava olioperustainen ohjelmointikieli, jonka tunnetuin käyttökohde on toimia skriptikielenä web-sivuilla. Se suoritetaan päätelaitteessa ja sen avulla voidaan luoda muun muassa dynaamisuutta web-sivuille erilaisiin tapahtumiin perustuen. JavaScriptiä ei pidä sekoittaa Javaan, vaikka niiden nimet ja syntaksit muistuttavatkin toisiaan, vaan ne ovat kaksi täysin erilaista ohjelmointikieltä. [5] JavaScriptin yleistymiseen on vaikuttanut vahvasti internetin yleistyminen, sillä JavaScript-ohjelmia voi upottaa HTML-dokumentteihin [35]. Koska JavaScript on ohjelmointikieli, sitä voidaan käyttää myös palvelinpään toteutuksessa. Eräs tunnettu palvelinpään ajoympäristö on JavaScript-tulkin sisältävä Node.js [40].

Termillä AJAX (Asynchronous JavaScript And XML) tarkoitetaan asiakkaan ja palvelimen välisen kommunikoinnin ja käyttöliittymän päivittämisen erottamista toisistaan. AJAX-termin mukaisesti näitä suoritetaan rinnakkain. Käyttäjän käyttöliittymä voi päivittyä heti, vaikka palvelimen vastaus vielä puuttuisikin. Perinteisissä staattisissa HTML-sivuissa selaimen pitää odottaa palvelimen vastausta ennen kuin se voi päivittää käyttöliittymää. AJAX-termin XML-osa on peräisin historiasta, jossa asiakkaan ja palvelimen väliseen kommunikointiin käytettiin XML-muotoa. [43] Nykyisin siirrettävän tiedoston muoto on tyypillisesti JSON (JavaScript Object Notation), joka on edelleen tekstipohjainen, mutta selkeästi tiiviimpi XML-muotoon verrattuna. AJAX:n avulla verkkosivustoista voidaan tehdä vuorovaikutteisia ja dynaamisia. Sisältö voidaan ladata pienemmissä osissa ja toisaalta tallennus voidaan tehdä taustalla.

Nykyisin on suosiossa niin sanotut yhden sivun sovellukset (SPA, Single-Page Application), joissa sivusto tai sovellus toimii yhden HTML-sivun kautta. Toiminta perustuu hyvin yksinkertaisen staattisen HTML-sivun lisäksi JavaScript-ohjelmaan, jonka avulla sivusto

toimii dynaamisesti. Mikäli SPA-sovellus sisältää dynaamisesti ladattavaa tai tallennettavaa sisältöä, se täyttää myös AJAX:n määritelmän. Näitä käytetäänkin monesti yhdessä. Sovelluksen suoritus hajautuu palvelimelta päätelaitteisiin, mikä osaltaan vähentää palvelimen kuormaa. Suosittuja SPA-sovelluskehyksiä ovat muun muassa Angular [6] ja React [47].

4.3.4 REST

REST (REpresentational State Transfer) on Roy Fieldingin vuonna 2000 esittelemä idea arkkitehtuurista, joka perustuu HTTP-protokollan metodien ja URL:n (Uniform Resource Locator) hyödyntämiseen hajautetuissa hypermediajärjestelmissä. REST toimii ulkoisena rajapintajana web-palvelulle, joka tyypillisesti tarjoaa mahdollisuuden vähintään tiedon luomiseen, hakemiseen, päivittämiseen ja poistamiseen. [17] URL kertoo palvelun tai tiedon sijainnin. URL ja HTTP-metodi yhdessä muodostavat semantiikan, jota REST:ssä hyödynnetään. Esimerkiksi GET-metodi kuvitteelliseen osoitteeseen `http://user.service/userinfo/4` voisi tarkoittaa käyttäjätietojen hakemista. Puolestaan PUT-metodikutsu samaan osoitteeseen voisi tarkoittaa käyttäjätietojen päivittämistä.

4.3.5 WebSockets

HTTP-liikenne on yhdensuuntainen. Asiakas ottaa yhteyden palvelimeen, joka palauttaa vastauksen. Kahdensuuntaiseen kommunikointiin ei ole ollut helppoa tapaa. Käytännössä se on toteutettu asiakaspään tekemillä jatkuvilla kyselyillä (*poll*). Tämä tekniikka on huono suorituskyvyn kannalta ja monimutkaistaa palvelimen toimintaa, koska molempiin suuntiin tarvitaan asiakaskohtainen TCP-yhteys. Ratkaisuna tähän on kehitetty WebSockets-teknologia, jonka avulla kahdensuuntainen kommunikointi on mahdollista yhden TCP-yhteyden päällä. WebSockets-teknologiaa voidaan soveltaa esimerkiksi pikaviestisovelluksissa, peleissä ja samanaikaisissa dokumenttien muokkauksissa. [23]

4.3.6 Java EE

Java EE (Java Enterprise Edition) on kokoelma web-kehitykseen liittyviä määrittelyjä ja standardeja, joiden avulla kuvataan, miten Java-web-ohjelmia voidaan tehdä. Java EE -ohjelmat ajetaan sovelluspalvelimessa (*application server*), joka tarjoaa määrittelyjä vastaavat toteutukset. [10] Versiosta 9 lähtien kehitysvastuu on ollut Eclipse Foundationilla ja nimi on muuttunut Jakarta EE:ksi. Tässä työssä käytetään vanhempaa nimeä sen ollessa huomattavasti tunnetumpi.

Sovelluspalvelin tarjoaa esimerkiksi web-palvelimen sisältäen istunnon, autentikoinnin, pääsynvalvonnan, tietokantayhteydet, riippuvuuksien injektion (CDI, Contextual Dependency Injection) ja transaktionhallinnan, jotka konfiguroidaan Java EE -sovelluksen tai sovelluspalvelimen konfiguraatiotiedoston avulla. Sovelluspalvelimia on hyvin eri tasoisia ominaisuuksin mitattuna. Java EE -referenssitoteutus on JBossin seuraaja WildFly [57], joka tarjoaa lähes kaikki Java EE -määrittelyjä vastaavat toteutukset. Toisena ääripäänä on Apache Tomcat [8], joka tarjoaa lähinnä HTTP-palvelimen ja Javalla tehtyjen web-sivujen esittämisen.

5. TUTKIMUSMENETELMÄT

Esimerkkitapausta arvioidaan tapaustutkimuksen tavoin. Tapaustutkimuksella ei ole tarkkaan määriteltyä kuvausta. Erään kuvauksen [44] mukaan sillä tarkoitetaan yhden tai useamman tapauksen käsittelyä. Se kuvaa lähestymistapaa selvitettävään asiaan. Tavoitteena on määritellä, analysoida ja ratkaista tarkasteltava tapaus. Ratkaisulla tarkoitetaan tutkimuskysymykseen tai -kysymyksiin vastaamista. Tapaustutkimus voi sisältää haastatteluja, huomioiden tekemistä ja muunlaisia tutkimuksia. Tutkimuksen arvioinnissa korostuvat laadulliset seikat. [44] Tämä diplomityö keskittyy yhteen tapaukseen, jota pyritään kuvaamaan mahdollisimman tarkasti, selvitetään siinä ilmenneitä seikkoja systemaattisesti ja mahdollisimman objektiivisesti sekä arvioidaan lopputulosta tutkimustulosten perusteella.

Uudistuksen arviointi perustuu erilaisiin mittauksiin. Mittaustavat voidaan jakaa kahteen pääkategoriaan; kvantitatiiviseen ja kvalitatiiviseen analyysiin. Ensimmäinen on niin sanottu määrällinen analyysi, jossa mitataan absoluuttisia arvoja esimerkiksi ohjelmistojen tapauksessa suoritusaikaa, datan määrää ja koodirivien lukumäärää. Jälkimmäinen, kvantitatiivinen analyysi kuvaa laadullista tarkastelua. Siihen liittyvät esimerkiksi haastattelut. Tässä työssä yhdistetään molempia mittausmenetelmiä.

5.1 Ohjelmiston laatu

Ohjelmistotuotteen tärkein mitattava asia on laatu. Laatua tarkastellaan ohjelmiston käyttäjän, tilaajan ja toimittajan näkökulmasta. Loppukäyttäjälle laadukkuus näkyy ohjelman oikeana toimintana. Tilaajan ja toimittajan näkökulmasta laadukas ohjelma on myös helposti ylläpidettävä, jotta mahdolliset myöhemmät muutokset voidaan tehdä vaivattomasti.

Ohjelmiston laatua varten on laadittu ISO 9126 -standardi. Sen tavoitteena on yhtenäistää ohjelmiston laadun arviointia määrittelemällä termistöä ja arvioitavia ominaisuuksia. Ohjelmiston laatu on jaettu useisiin eri kategorioihin, jotka jaetaan pienempiin osatekijöihin. Esimerkiksi ylläpidettävyyteen vaikuttavat analysoitavuus, muutettavuus, stabiilisuus, testattavuus ja ylläpidettävyyden helppous. [1] Myöhemmin ISO 9126 on uudistunut ISO/IEC 25000 -standardiperheeksi, joka kattaa laadun määrittämiseen, mittaamiseen ja evaluointiin liittyviä yhtenäisiä käytäntöjä ja ohjeita [24]. Laatupiirteet ovat aina tapauskohtaisia ja standardista pitääkin valita kuhunkin tapaukseen soveltuvat mittarit.

Ohjelmiston laadun mittaamisessa mittaustavat ja konkreettiset mitattavat asiat ovat tärkeässä roolissa. Haikala ja Märijärvi esittävät kirjassaan Ohjelmistotuotanto eräiksi objektiivisiksi tuotteen laadun mittareiksi ohjelmiston koon, virheiden määrän ohjelmakoodinriviä kohti, mutkikkuusmitat, palveluasteen, vikaantumisvälin, korjausajan, erilaiset tehokkuusmitat sekä asiakastyytyväisyyden [20]. Laatua tarkastellaan näiden mittareiden pohjalta. Mittareista jätetään pois sellaiset, jotka eivät sovellu tarkasteltavaan tapaukseen.

Ohjelmiston koko on hyvä kvantitatiivinen mittari, koska sen voi mitata nopeasti, helposti ja melko yksikäsitteisesti. Ohjelmiston kokoa voidaan arvioida koodirivien, funktioiden, moduulien ja luokkien määrän avulla sekä yksittäisten tiedostojen määrän perusteella. Eri kokoisia ohjelmia voidaan verrata keskenään, mikäli muut mitatut laatuominaisuudet suhteutetaan ohjelmiston koon mukaan. Esimerkkitapauksessa käytetään mittana tiedostojen ja koodirivien määrää. Tiedostojen määrää voidaan käyttää mittarina, koska Javan käytäntöjen mukaan jokainen luokka kirjoitetaan omaan tiedostoon. Tällä on löyhä assosiaatio Clojuren nimiavaruuksiin, joista jokainen kirjoitetaan erilliseen tiedostoon. Koodimäärän luotettavuutta mittarina heikentää lukuisat ulkoiset kolmansien osapuolten tekemät kirjastot ja sovelluskehykset, joita hyödynnetään ohjelmakoodissa. Osa toiminnallisuutta tukevasta toteutuksesta on tehty ulkoisiin kirjastoihin, joita ei lasketa koodimäärään mukaan. Toisaalta näiden kirjastojen lukumäärä voidaan huomioida mittauksessa. Funktioiden, moduulien ja luokkien vertaileminen ei anna luotettavia tuloksia, koska ohjelmointikielet ovat niin erilaisia.

Ohjelmiston mutkikkuus eli monimutkaisuus kertoo, kuinka helppoa siihen on tehdä muutoksia tai lisätä uusia ominaisuuksia. Mutkikkaan ohjelman ymmärtäminen on vaikeaa. Tavoitteena on tehdä ohjelmista mahdollisimman yksinkertaisia, jotka kuitenkin toteuttavat niille asetetut vaatimukset. Mutkikkuuden mittaamiseen käytetyt mitat riippuvat suuresti ohjelmointiparadigmasta. Olio-ohjelmoinnissa monimutkaisuutta voi mitata esimerkiksi luokan metodien kompleksisuuden avulla, perintäpuun syvyydellä ja olioiden välisen sidonnan perusteella [51]. Nämä eivät sovellu sellaisenaan funktionaaliseen kieleen, koska olioita ei ole. Mutkikkuus vaikuttaa kuitenkin oleellisesti ohjelman ymmärrettävyyteen ja tätä kautta ylläpidettävyyteen. Mutkikkuusmittojen hyödyllisyydestä on ristiriitaisia mielipiteitä, sillä muun muassa monimutkainen ohjelma on mitattuna myös monimutkainen ja eri arvojen vertailu keskenään on vaikeaa [21].

Harsu kertoo kirjassaan, että kytkentämitat *(coupling metrics*) kuvastavat, paljonko ohjelmalla tai moduulilla on kytkentöjä ulkopuolelle. Suuri lähtevien kutsujen määrä tarkoittaa, että aliohjelma on todennäköisesti monimutkainen ja tätä kautta sillä on heikompi ylläpidettävyys ja uudelleenkäytettävyys. [34]

Ohjelmiston mutkikkuutta arvioidaan valitusta kohdasta ohjelmakoodia. Kohdaksi valitaan ohjelmassa oleva tyypillinen toiminnallisuutta sisältävä ohjelmakoodilohko. Valitun kohdan sisältämä logiikka ja toiminnallisuus toistuvat varioituna useammassa kohdassa ohjelmakoodin sisällä, minkä perusteella katetaan suurin osa sovelluksen ymmärtämiseen vaikuttavista toimintasäännöistä ja sovelluslogiikasta eikä laajempi tarkastelu ole tarpeellista. Kohdasta tarkastellaan lähtevien kytkentöjen määrää moduuleihin, jotka sisältävät toiminnallisuutta. Toiminnallisuutta sisältäviksi moduuleiksi lasketaan Java-puolella injektiopisteet tarkasteltavassa luokassa sekä kaikissa sen kantaluokissa. Injektiopisteillä tarkoitetaan sovelluspalvelimen tarjoamaa mahdollisuutta asettaa moduulien ja palveluiden riippuvuudet automaattisesti tai konfiguraatiotiedostojen mukaisesti ohjelmakoodiin staattisesti asetettujen riippuvuuksien sijaan. Funktionaalisella puolella riippuvuuksiksi lasketaan sivuvaikutuksia sisältävän moduulit. Uudistettua sovellusta vertaillaan aikaisempaan toteutukseen.

Ohjelman alkuperäisenä vaatimuksena on 100 %:n palveluaste, joka tarkoittaa, että ohjelma on aina päällä ja käytettävissä. Palveluastetta vähentää esimerkiksi virhetilanteista toipumiseen kuluva aika, jolloin järjestelmä ei ole käytettävissä. Tämä laadun mittari jätetään huomioimatta, koska suuri palveluaste on järjestelmän vaatimuksena.

Esimerkkitapauksessa vikaantumisväli ei sovellu arviointiin, koska suurin osa ohjelman suorituksesta on lähtöisin käyttäjästä ja tätä kautta ohjelman vikatilanteet ovat suoraan verrannollisia käyttäjämäärään ja käytön määrään. Käyttäjämäärän mukaan vakioitua vikatilanteiden ilmaantumista voidaan mitata laskemalla vikatilanteet käyttäjämäärää kohti. Ohjelman oletetaan aina toimivan ja vikaantumisväliä parempi mittari onkin virheiden määrä. Virheiden määrän ja vikaantumisvälin luotettava käyttö mitattavina suureina edellyttää pidempää tuotantokäyttöä, joka ei tämän työn puitteissa ole mahdollista. Jotta raportoituja virheitä voisi käyttää luotettavana mittarina, ne on myös todennettava oikeiksi ohjelmistovirheiksi. Virheisiin voi helposti sekoittua väärinymmärrykset vaatimusmäärittelyssä tai myöhemmin halutut muutokset alkuperäisiin vaatimuksiin nähden.

Asiakastyytyväisyyden avulla on mahdollista arvioida uudistettua ohjelmistoa varsinkin tilanteessa, jossa vanha ohjelmisto korvataan uudella. Loppukäyttäjille käyttöliittymä on hyvin samankaltainen kuin aikaisemmin, mutta ei-toiminnallisissa ominaisuuksissa, kuten suorituskyvyssä, tavoitellaan parantumista. Asiakastyytyväisyys olisi hyödyllisintä mitata pitkän aikavälin kuluessa. Tähän ei kuitenkaan ole tämän tutkimuksen puitteissa mahdollisuutta.

Laadun tarkasteluun on kehitetty useita erilaisia työkaluja kuten staattiset koodianalysaattorit ja ohjelman suoritusta analysoivat profilointiohjelmat. Esimerkkinä mainittakoon

McCaben monimutkaisuutta [54] mittaavat työkalut ja suorituskykyä mittaava Apache JMeter [7]. Ohjelmointiparadigman ja -kielen muutos vaikeuttaa näiden työkalujen käyttöä sillä saadut tulokset eivät ole keskenään vertailukelpoisia. Lisäksi Clojurelle ei ole olemassa yhtä kattavaa ja vakiintunutta työkalujen joukkoa laadun testaamiseen kuin Javalle.

Ohjelmiston laatua ja parantumista mitataan soveltuvin mittarein. Nämä mittarit ovat ohjelmiston koko koodirivien ja kooditiedostojen määrän mukaan mitattuna ja ohjelman teknisen rakenteen yksinkertaisuus riippuvuuksin mitattuna. Varsinaisen ohjelmakoodin lisäksi sovelluksen kokoa ja monimutkaisuutta mitataan tietokannan taulumäärän perusteella.

5.2 Haastattelututkimus

Kvalitatiivisia eli laadullisia tuloksia selvitetään haastattelun avulla. Haastattelututkimuksessa pyritään saamaan tietoa niistä asioista, joita ei voi suoraan mitata kvantitatiivisella analyysilla. Haastattelututkimuksen avulla on vaikea saada objektiivisia tuloksia, jotka ovat lähtökohtana ohjelmiston mittaamiselle.

Haastattelututkimukset voidaan jakaa lomakehaastatteluihin, teemahaastatteluihin ja strukturoimattomiin haastatteluihin. Lomakehaastattelu tehdään nimensä mukaisesti lomakkeen mukaan. Lomakkeen kysymysten ja väitteiden muoto, esitysjärjestys ja mahdolliset vastausvaihtoehdot ovat ennalta määrätyt. Etuna on haastattelun tekemisen helppous. Vaikeutena on kysymysten laatiminen yksikäsitteisesti. [52]

Strukturoimaton haastattelu tehdään ilman ennalta määrättyä rakennetta. Se muistuttaa paljon avointa keskustelua, jossa haastattelija pyrkii syventämään haastateltavien vastauksia ja rakentamaan haastattelun jatkon niihin pohjautuen. Haastateltaviksi valitaan erikoistuneita henkilöitä. [52]

Lomakehaastattelun ja strukturoimattoman haastattelun välimaastoon jää teemahaastattelu. Haastattelijalla on ennalta laaditut kysymykset, mutta hän voi vaihdella niiden järjestystä tai muuttaa sanamuotoja. Haastateltavat voivat vastata omin sanoin eivätkä vastaukset ole sidottuja vastausvaihtoehtoihin. Teemahaastattelussa jokin haastattelun näkökulma on lyöty lukkoon, mutta muuten haastattelu on melko vapaamuotoinen. [52]

Tämän työn onnistumisen arviointia tehdään teemahaastattelun avulla. Haastateltaviksi valitaan asiantuntijoita, jotka tuntevat ohjelman teknistä toteutusta sekä asiakasnäkökulmaa ja projektinjohdollista näkökulmaa. Teemahaastattelua varten laaditaan runko, jonka pohjalta haastattelut käydään. Haastattelun tarkoituksena on selvittää ohjelmiston uudistamiseen liittyviä laadullisia seikkoja ja niissä onnistumista. Ohjelmistokehittäjiltä

selvitetään kysymyksien perusteella uudistuksen onnistumisen lisäksi uudistusprojektin aikana tehtyjä valintoja sekä pyritään ohjaamaan pohdintaa myös hieman laajempaan näkökulmaan kuin vain tehtyyn uudistusprojektiin. Teemahaastatteluun päädyttiin, koska lomakehaastattelua ei ole mielekästä tehdä haastateltavien pienen määrän vuoksi. Lopputulosten arvioitiin olevan parempia kuin kokonaan strukturoimattomalla haastattelulla.

5.3 Haastattelukysymykset

Teemahaastattelu tehdään projektissa mukana olleille sovelluskehittäjille ja tuoteomistajalle. Teemahaastattelun esitehtävänä on vastata kyselyyn. Kysely koostuu monivalintakysymyksistä, joiden avulla johdatetaan varsinaisessa haastattelussa käytäviin aihealueisiin. Esitehtävän väittämät on esitetty taulukossa 2. Mielipideväittämissä käytetään tavallisesti Likertin asteikkoa, joka jakaantuu 4- tai 5-portaiseen järjestysasteikkoon [53]. Likertin asteikon mukaisesti kyselyn väittämiin on mahdollista vastata asteikolla yhdestä viiteen, jossa yksi tarkoittaa, että on täysin eri mieltä väittämän kanssa, ja viisi tarkoittaa, että on täysin samaa mieltä.

Taulukko 2 *Kyselyn väittämät*

Järjestys-numero	Väittämä
1.	Uuden ominaisuuden (toimenpiteen) tekeminen ja toteutus on helpompaa kuin vanhassa.
2.	Uudistettuun sovellukseen on helppo tehdä muutoksia.
3.	Ohjelmistovirheen korjaus on helpompaa uudistettuun sovellukseen.
4.	Virheellisen datan korjaus on helpompaa uudistuksen jälkeen.
5.	Suorituskyky on parempi uudistuksen jälkeen.
6.	Sovellusta voidaan hyödyntää toisissa vastaavissa järjestelmissä.
7.	Uudistettu sovellus on laadukkaampi kuin edeltävä.
8.	Funktionaalisella paradigmalla on saavutettu selkeitä hyötyjä.
9.	Uudistus on onnistunut.

Esitehtävänä toimineen kyselyn jatkona pidetään haastattelu. Haastattelukysymyksien avulla pohditaan vanhan sovelluksen heikkouksia ja vahvuuksia suhteessa uudistettuun sovellukseen. Lisäksi tarkastellaan hankaliksi osoittautuneita osa-alueita, joiden perus-

teella myös uudistuksen alkuperäiset tavoitteet on laadittu. Kysymysten asettelu on mahdollisimman objektiivinen. Haastateltavia ei johdatella tiettyihin näkökulmiin, vaan heidän mielipiteensä pohjautuvat omiin näkemyksiinsä.

Varsinaiset haastattelukysymykset ovat listattuna taulukossa 3. Harmaalla taustalla olevat kysymykset on tarkoitettu vain sovelluskehittäjille, koska kysymykset ovat luonteeltaan teknisempiä ja tuoteomistajan on lähes mahdotonta vastata näihin itsenäisesti omaan arvioon pohjautuen. Kaikille annetaan kuitenkin mahdollisuus kommentoida kysymyksiin liittyviä aihealueita sekä muita mieleen tulevia asioita. Haastattelu tehdään kunkin asiantuntijan kanssa kahdenkeskisesti. Haastattelut noudattavat tiettyä runkoa, mutta siitä voidaan lievästi poiketa tarkentavilla kysymyksillä ja haastateltavien kertoessa muita uudistusprojektin arvioinnin kannalta hyödyllisiä tietoja.

Taulukko 3 *Haastattelukysymykset ja -runko*

Järjestys-numero	Kysymys
1.	Minkälainen on yleistuntemus uudistuksen tekemisen ajalta?
2.	Mitä hyviä puolia näet uudistetussa sovelluksessa verrattuna vanhaan?
3.	Mitä hyviä puolia näet vanhassa sovelluksessa verrattuna uudistettuun?
4.	Kuinka joustava uudistettu järjestelmä on uusien muutosten varalta?
5.	Miten arvioit ohjelmistovirheen korjauksen tekemistä verrattuna vanhaan järjestelmään?
6.	Miten virheellisen datan korjaaminen tietokantaan eroavat uudessa ja vanhassa järjestelmässä?
7.	Miten uusi ja vanha järjestelmä eroavat suorituskyvyssä?
8.	Voidaanko järjestelmää hyödyntää kokonaan tai osittain uudelleen toisessa projektissa?
9.	Miten arvioit uudistetun ohjelmiston laatua verrattuna vanhaan?
10.	Onko funktionaalisesta kielestä ollut hyötyä uudistamisen yhteydessä?
11.	Onko uudistus onnistunut? Mikä uudistuksessa on mennyt hyvin ja missä asioissa olisi ollut parannettavaa?

Kukin haastattelu dokumentoidaan myöhempää tiivistelmää ja yhteenvetoa varten. Haastatteluista kerätään uudistamiseen liittyvät ilmi tulleet asiat ja mielipiteet sekä haastateltavien taustat. Haastatteludokumentaation pohjalta koostetaan ilmenneet asiat,

jotka toistuvat useamman haastateltavan välillä. Lisäksi kerätään kunkin haastateltavan erikseen painottamat asiat. Aineiston lopputuloksesta poistetaan vähemmälle jääneet ja vain yksittäisiä mainintoja saaneet asiat.

5.4 Arviointi

Ohjelmistoprojektien konkreettisia hyötyjä saattaa olla hankala mitata varsinkin edeltävän toteutuksen korvaamisessa. Tuloksien perusteella kuitenkin pyritään selvittämään, kuinka onnistunut uudistus oli kokonaisuudessaan, missä asioissa onnistuttiin ja missä jäi parantamisen varaa.

Kokonaisuuden onnistumista arvioidaan kategorioittain, joista muodostetaan kokonaiskuva. Kategorioiden perusteella arvioidaan myös yksittäisten osa-alueiden onnistumista. Kategoriat on kuvattu taulukossa 4. Osa-alueet on valittu ohjelmistoarkkitehtuurien arvioinnin perusteiden mukaan [30], joista on poistettu sellaiset osa-alueet, jotka eivät sovellu tarkasteltavaan sovellukseen. Osa-alueisiin on lisätty uudistettavan sovelluksen ylläpidossa hankaliksi koettuja ja havaittuja asioita.

Taulukko 4 *Uudistamisen arvioinnin osa-alueet*

Osa-alue	Selite
Ylläpidettävyys, ohjelmistovirheen korjaus	Ohjelma toimii virheellisesti ja siihen tehdään korjaava muutos. Kuinka nopeaa ja helppoa muutoksen tekeminen on?
Ylläpidettävyys, virheellisen datan korjaus	Tuotantoon on tallentunut virheellistä dataa ja se pitää korjata jälkikäteen. Kuinka nopeaa ja helppoa datan korjaus on?
Jatkokehitettävyys	Ohjelmaan halutaan uusi toiminnallisuus. Kuinka helposti se onnistuu?
Ylläpidettävyys, joustavuus	Nähtävillä olevien tulevien muutosten tekemisen helppous.
Tehokkuus	Kuinka suorituskykyinen järjestelmä on verrattuna aikaisempaan toteutukseen?
Uudelleenkäytettävyys	Voidaanko ohjelmaan tehdä uusia ominaisuuksia? Voidaanko ohjelmaa hyödyntää muussa samaan aihealueeseen liittyvässä asiassa?
Laatu	Onko lopputulos laadukkaampi kuin aikaisempi toteutus?
Kokonaisuus	Onnistumisen arviointi kokonaisuudessaan.

Haastattelututkimuksen kysymykset jaetaan taulukossa 4 esitettyihin osa-alueisiin. Kysymyksien avulla arvioidaan kunkin osa-alueen onnistuminen erikseen. Haastatteluun kuuluvat myös tarkentavat kysymykset, joilla pyritään saamaan haastateltavat pohtimaan asioita syvemmällä tasolla. Myös ohjelmiston laadun mittarit jaetaan näihin osa-alueisiin ja kokonaisuuden onnistumista arvioidaan niiden avulla.

Haasteena kyselyssä ja haastattelussa tulee olemaan pieni otosmäärä. Osallistuvia henkilöitä on viitisen kappaletta. Rajoite tulee suoraan esimerkkiprojektiin osallistuneiden henkilöiden lukumäärän perusteella. Tätä pientä lukumäärää pyritään kompensoimaan haastatteluiden laadulla. Haastattelut pidetään henkilökohtaisesti ja ilman tiukkaa aikarajoitusta. Tulosten arvioinnissa suurin painoarvo on haastattelututkimuksella ja ohjelmiston laadun metriikoilla.

6. ESIMERKKITAPAUS

Ohjelmiston uudistamista ja funktionaalisen ohjelmointiparadigman vaikutusta tarkastellaan Netum Oy:ssä tehdyn projektin avulla. Tässä luvussa kuvataan uudistettava ohjelmisto, uudistukseen johtaneet syyt ja kerrotaan uudistuksen tekemisestä.

6.1 Ohjelmiston kuvaus

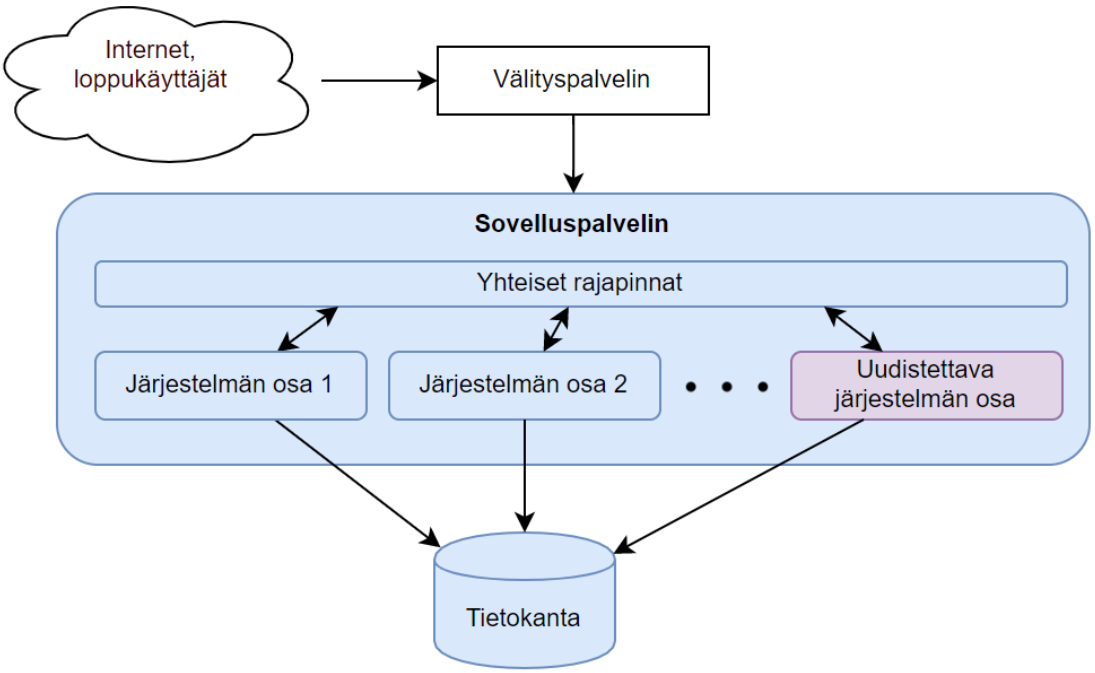
Uudistusprojektin tavoitteena on uudistaa Netum Oy:n asiakasprojektin ohjelmistosovellus. Tämä web-sovellus toimii julkisessa internetissä. Uudistus koskee erästä loogista osakokonaisuutta. Uudistettavan osan toiminnallisuuden voi mieltää kolmeksi rinnakkaiseksi hierarkkiseksi puurakenteeksi, joiden lehtien välillä tapahtuu rahatapahtumia. Näitä tapahtumia syntyy toimintaprosessien lopputuloksena. Yksittäiset rahatapahtumat liittyvät aina kahteen saman puun lehteen. Jokaiseen rahatapahtumaan liittyy yksi tai useampi lähde ja kohde sekä toimintaprosessi, joka pitää sisällään täytettäviä lomakkeita, useita eri käyttäjäroolein eroteltuja yksittäisiä toimintoja ja lopputuloksina syntyviä arkistoitavia PDF-dokumentteja. Ohjelman toiminta on pääpiirteittäin melko yksinkertainen, mutta monimutkaisuutta lisäävät useat yksityiskohdat ja poikkeustapaukset. Tarkempi toiminnallinen kuvaus ei ole tarpeellinen uudistuksen analysoinnissa. Oleellisempaa on tietää ohjelman tekninen rakenne, käytetyt teknologiat ja arkkitehtuuritason suunnitteluperiaatteet.

Alkuperäisen ohjelman toteutus on aloitettu vuonna 2013. Uudistettava osuus on pääpiirteittäin valmistunut vuoteen 2016 mennessä. Tuotantokäytössä järjestelmä on ollut vuodesta 2014. Työmääränä mitattuna uudistettavan järjestelmän koko on ollut noin 12 henkilötyövuotta. Ohjelman uudistuksen kannalta on hankalaa, että alkuperäisiä kehittäjiä ei ole saatavilla uudistusprosessiin.

Kokonaisjärjestelmä on suurehko pääosin Java-kielellä tehty Java EE -sovellus, jonka eräs hankalimmin ylläpidettävä osa korvataan uudella toteutuksella. Olemassa olevien ominaisuuksien muuttaminen ja uusien tekeminen on ollut selkeästi eniten aikaa vievää tässä osassa muihin osiin verrattuna. Lisäksi siihen on kohdistumassa sellaisia toiminnallisia muutoksia, jotka edellyttävät suuria teknisiä muutoksia. Tietorakenteita yksinkertaistetaan ja sovelluslogiikan ydin eriytetään toiminnallisesta logiikasta mahdollisimman hyvin. Sisäiseltä arkkitehtuuriltaan uudistettava osa noudattaa kerrosarkkitehtuuria ja tarkemmin kuvattuna MVC-mallia (*Model, View, Control* – Malli, Näkymä, Kontrolli), jossa

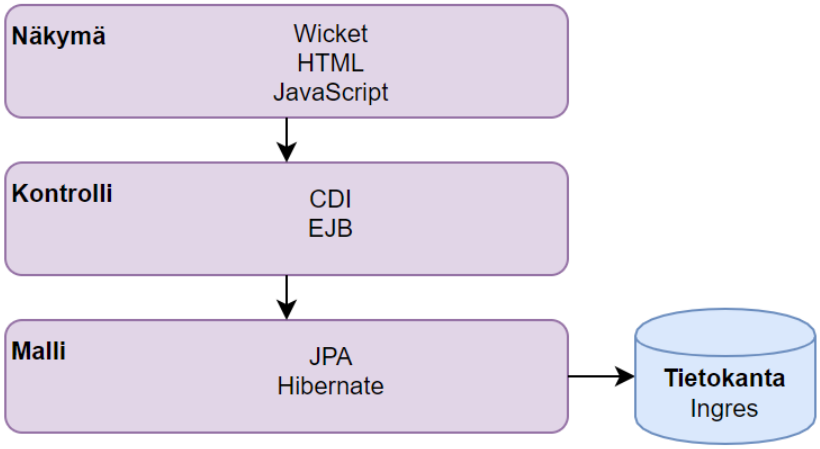
jokainen kerros toimii mahdollisimman itsenäisesti ja kerrosten väliset kutsut ovat hallittuja, tyypillisesti ylhäältä alaspäin. Uudistettava osa on kiinteästi kytköksissä muuhun sovellukseen ohjelmakoodin tasolla. Kokonaisuutta suoritetaan sovelluspalvelimessa. Alkuperäinen kokonaisuus noudattaa monoliittiarkkitehtuuria, joka koostuu pienemmistä osista.

Järjestelmäkokonaisuuden sovellusarkkitehtuuri on esitetty kuvassa 2. Kokonaisuus koostuu useista osamoduuleista, jotka ovat yhteydessä toisiinsa yhteisten rajapintojen kautta. Jokainen osa noudattaa sisäisesti MVC-arkkitehtuuria. Kokonaisuuden kaikki loogiset osat käyttävät samaa tietokantaa. Osioiden omat taulut on erotettu käyttämällä etuliitteitä taulujen nimissä. Loppukäyttäjät käyttävät järjestelmää välityspalvelimen kautta, joka ohjaa liikenteen sovelluspalvelimeen.



***Kuva 2** Järjestelmäkokonaisuus ja uudistettava osa*

Uudistettava järjestelmän osa on MVC-mallin mukaisesti jaettu kolmeen eri kerrokseen, joiden välillä kommunikointi tapahtuu ylhäältä alaspäin. Osan sisäinen tekninen arkkitehtuuri ja keskeisimmät käytetyt teknologiat on esitetty kuvassa 3. Selkeyden vuoksi kuvasta on jätetty pois kerrosten väliseen kommunikointiin käytetyt luokat. Lisäksi ulkopuolelle on jätetty liityntäpisteet muihin kokonaisjärjestelmän osiin.



Kuva 3 *Uudistettavan sovelluksen sisäinen rakenne ja käytetyt teknologiat*

Uudistettavan osan näkymäkerroksessa on käytetty Apache Wicket-teknologiaa [9], joka on avoimen lähdekoodin komponenttipohjainen sovelluskehys (*application framework*). Wicket hoitaa käyttöliittymän tilan käsittelyn ja kommunikoinnin palvelimen ja asiakkaan välillä. Kehittäjän tarvitsee osata vain HTML- ja Java-kieliä. Teknologia perustuu dynaamisesti muuttuviin HTML-sivuihin, jossa muutokset tapahtuvat hyödyntäen HTTP:n lisäksi AJAX-teknologiaa. Palvelin muodostaa vastaukset ja kommunikoinnissa välitetään valmista HTML-merkkauskieltä, jonka selainohjelma esittää käyttäjälle.

Kontrollikerros sisältää ohjelman sovelluslogiikan. Tämä on kooltaan ehdottomasti suurin ja monimutkaisin osa. Uudistettavaan ohjelmaan kuuluu noin 10 eri tyyppistä prosessia, joiden toimintalogiikka, käyttöoikeudet ja toimintasäännöt on kuvattu kontrollikerroksessa. Kokonaisjärjestelmälle tarjottavien sovellusten sisäisten rajapintojen toteutus kattaa merkittävän osan kontrollikerroksen sisällöstä. Käytettyjen CDI- ja EJB-teknologioiden avulla on hoidettu pääasiassa riippuvuuksien ja transaktioiden hallinta.

Malli-kerros hoitaa tietokantayhteyksiä ja datan tallennuksen pysyväismuistiin. Sovelluksessa on käytetty tiedon varastoinnissa ORM-työkaluna (Object Relational Mapping) JPA:n (Java Persistence API) ja Hibernaten yhdistelmää. Työkalujen avulla kuvataan ohjelmakoodissa käytetyt objektit tietokannan tietorakenteiksi. Näiden avulla sovelluskehittäjän ei tarvitse erikseen huolehtia tietokannan rakenteesta, vaan eri luokkien väliset relaatiot on kuvattava sen sijaan ohjelmakoodissa.

Uudistettavan ohjelman koodirivien määrä on eroteltu ohjelmointikielikohtaisesti taulukossa 5. Suurin osa ohjelmakoodista on Java-kieltä. Toisena tulee SQL, jonka suuri rivimäärä selittyy Java-luokista generoidulla kannan rakenteen kuvauksella. Tällä ei kuiten-

kaan ole merkitystä ohjelman suorituksen aikana. Kolmantena on HTML, joka liittyy käyttöliittymään ja Wicketin käyttöön. Neljäntenä on XML, joka koostuu lähinnä Apache Maven -paketointi- ja -koontityökalun konfigurointitiedostoista. Viimeisenä on LESS, jota käytetään CSS-koodin generointiin.

Taulukko 5 *Koodirivien määrä ohjelmointikielittäin eroteltuna*

Ohjelmointikieli	Koodirivien määrä tuhansina riveinä (KLOC)
Java	226,3
SQL	11,5
HTML	5,8
XML	2,6
LESS	1,0

6.2 Uudistuksen lähtökohta

Ohjelman kehitys ja ylläpito on käynyt hankalaksi, koska ohjelman tekninen rakenne on monimutkainen ja kankea. Luokkahierarkioita ja riippuvuuksia on käytetty tehokkaasti hyödyksi, mikä osaltaan vaikeuttaa oleellisen sovelluslogiikan ymmärtämistä. Kehittäjällä pitää olla mielessään useiden luokkien ja rajapintojen hierarkia ja keskinäiset suhteet, jotta ymmärtää sovelluksen ja voi tehdä halutun muutoksen oikeaan kohtaan. Käytännössä muutosten tekeminen on koettu hankalaksi.

Käytetyt teknologiat ovat vanhentuneet erityisesti käyttöliittymäkerroksessa. Siellä välitetään verkon yli HTML-merkkauskieltä. Nykyään suuntauksena on datan välittäminen esimerkiksi JSON-muodossa. Modernisoinnin hyötynä on ohjelman nopeampi toiminta ja pienempi tietoliikenneresurssien ja palvelimen laskentatehon käyttö. Eräs huomioitava käyttöliittymäteknologiaan liittyvä seikka on Wicket-osaajien pieni lukumäärä JavaScript-pohjaisia teknologioita osaaviin henkilöihin verrattuna.

Palvelinpuolella ajettava koodi on tehty Java-kielen versio 7:lla eikä version päivittäminen ole suoraviivaista. Päivittäminen uudempaan edellyttäisi sovelluspalvelimen päivityksen uudempaan, sovelluspalvelimeen tehtyjen muutosten tarkan analysoinnin, käyttöliittymäkirjaston päivittämisen ja ohjelmakoodin katselmoinnin kokonaisuudessaan muutosten varalta. Nykyään käytössä oleva Java 7 on julkaistu vuonna 2011 eikä siihen saa enää julkisia päivityksiä [28].

Tietokantaa käsitellään ORM-työkalun avulla. Tämä on johtanut nopeasti kehitettäviin, mutta tehottomiin tietokantakyselyihin. Ohjelma esimerkiksi summaa tietoja ohjelmakoodissa myös niissä tapauksissa, joissa laskentatoimet voitaisiin tehdä tietokantamoottorin avulla. Tämä on suorituskykymielessä hyvin hidasta, koska jokainen tietokannan taulun rivi kuvataan ensin Java-olioina, joihin perustuen laskutoimitukset tehdään. Tehokkainta olisi tehdä laskenta tietokannan avulla, jolloin kuvaus objekteiksi tapahtuisi vain lopputuloksesta.

Sovelluksen suorituskyvyn ylläpitäminen alkaa olla haastavaa. Sovellus on toiminut hyvin alkuvaiheessa, mutta ajan saatossa ja datamäärän lisääntyessä suorituskyky on laskenut eksponentiaalisesti. Alkuvaiheessa on selvitty kasvattamalla transaktion aikarajan pituutta. Nykyisellään on nähtävillä, että edes tämä ei ole riittävää järjestelmän jäljellä oleva elinaika huomioiden.

Ohjelman päivittäminen tuotantoon edellyttää aina koko sovelluksen päivittämistä, mikä aiheuttaa käyttökatkon. Osittainen päivittäminen olisi mahdollista, jos osa toimisi itsenäisesti eikä sillä olisi vahvaa riippuvuutta muuhun järjestelmään. Lisäksi kehittäminen olisi nopeampaa, koska ohjelmistokehittäjien ei tarvitsisi käsitellä kokonaisjärjestelmää kehitysympäristössään.

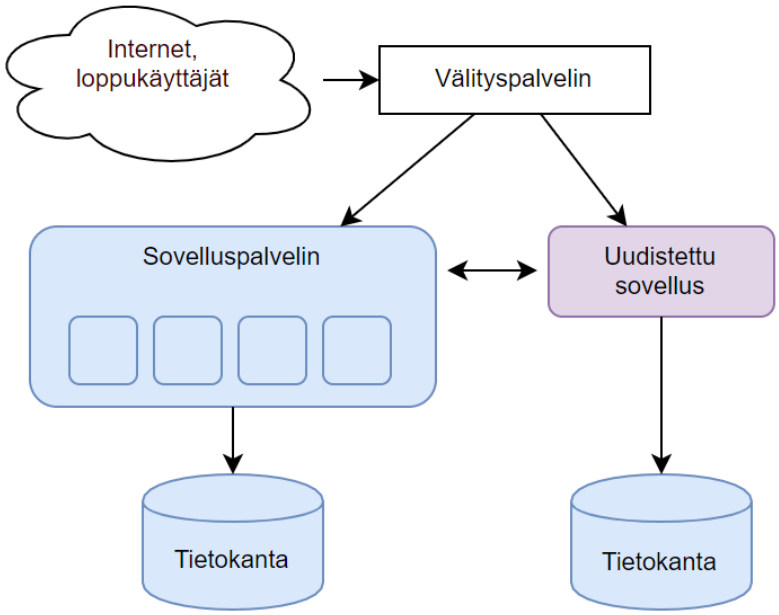
Uudistuksen tavoitteena on helpottaa ohjelman jatkokehitystä, parantaa sen uudelleenkäytettävyyttä ja ylläpitoa, modernisoida siinä käytettyjä teknologioita sekä parantaa suorituskykyä. Toiminnallisuuden selvittämiseen käytetään alkuperäisiä määrittelyjä ja tuoteomistajien haastatteluja. Uudistettavaan sovellukseen perehdytään lähdekoodin perusteella ja tutkimalla sen toiminnallisuutta käyttöliittymän kautta. Uudistuksessa huomioidaan sovelluksesta annettu aikaisempi palaute. Myös nykyisiä ylläpitäjiä kuullaan ja selvitetään ongelmakohdat ja hankalimmin ylläpidettävät asiat.

6.3 Uudistuksen toteutus

Uudistuksessa oli mukana alkuperäisen uudistettavan ohjelman toiminnallisuuden asiantuntijoita, joten uudistusta alettiin tehdä toiminnallisen määrittelyn pohjalta. Myös loppukäyttäjiä oli mukana, minkä ansiosta saatiin palautetta vanhasta järjestelmästä mahdollisimman suoraan. Toiminnallisen määrittelyn lisäksi teknisiä yksityiskohtia selvitettiin vanhasta ohjelmakoodista. Tämä oli kuitenkin melko pienessä roolissa ohjelmiston vaikeaselkoisuuden takia. Uusi muodostettu toiminnallinen määrittely ei ollut täysin yhteneväinen vanhan kanssa. Halutut toiminnalliset muutokset toteutettiin uudistuksen yhteydessä. Uudistus oli siis aikaisemman ohjelman korvaus, jossa käyttöliittymä, sisäiset tietorakenteet ja datamalli korvattiin uusilla.

Aliluvussa 2.3 esitettyyn uudistamisen prosessimalliin verrattuna esimerkkiprojektin uudistaminen perustui alkuperäisen arkkitehtuurimallin ymmärtämiseen ja analysointiin, mitä kautta uutta arkkitehtuurimallia alettiin rekonstruoida. Osa toiminnallisuudesta ja yksityiskohdista takaisinmallinnettiin osaksi alkuperäistä mallia. Uusi arkkitehtuurimalli perustui muuttuneisiin vaatimuksiin sekä parannettaviin ominaisuuksiin. Uudistettu sovellus toteutettiin muodostuneen uuden mallin perusteella.

Kokonaisjärjestelmään tehtiin arkkitehtuuritason muutos, jossa aikaisemmin yhtenä osana ollut alijärjestelmä irrotettiin kokonaan itsenäisesti toimivaksi järjestelmäksi. Uusi järjestelmäkokonaisuus ja uudistetun sovelluksen liittyminen muuhun järjestelmään on esitetty kuvassa 4. Järjestelmien erottamisella tavoiteltiin parempaa uudelleenkäytettävyyttä. Kokonaisjärjestelmän osaa voidaan uudelleen käyttää ilman mukana tulevaa muuta järjestelmää. Lisäksi pienemmän kokonaisuuden kehittäminen on nopeampaa käytännön työssä, koska muuta järjestelmää ei pääasiassa tarvitse käynnistää. Myös ohjelman ymmärtäminen ja omaksuminen helpottuu ohjelman pienentyessä.

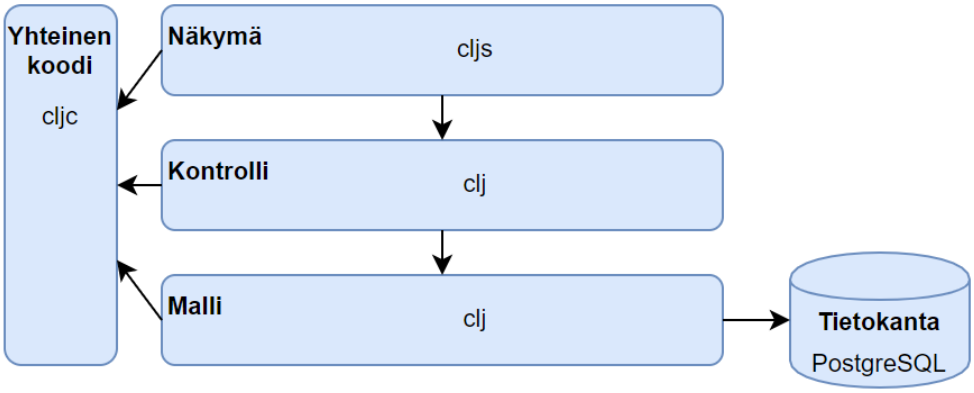


Kuva 4 *Uudistettu järjestelmäkokonaisuus*

Muun järjestelmän ja uudistetun sovelluksen välille jäi tarve kommunikoida keskenään. Osa uudistetun sovelluksen prosesseista käynnistyy vanhan järjestelmän kautta. Lisäksi järjestelmien välillä täytyy välittää tietoa sen hetkisestä tilasta. Järjestelmien rajapinta

hoidetaan REST-teknologialla, jonka käyttö eristetään palvelimien väliseksi. Muuhun sovellukseen oli jo aikaisemmin tehty REST-palveluita, joten niiden lisääminen oli suoraviivaista.

Uudistetun sovelluksen sisäinen rakenne noudattaa edelleen MVC-arkkitehtuuria ja on hyvin samanlainen kuin alkuperäisessä sovelluksessa. Alkuperäisen sovelluksen sisäinen arkkitehtuuri koettiin olevan toimiva, joten sitä ei lähdetty muuttamaan. Suurimpana erona on yhteisen koodin käyttö näkymä- ja kontrollikerroksessa. Uudistettu sovellus on esitetty kuvassa 5.



Kuva 5 *Uudistetun sovelluksen sisäinen rakenne*

6.3.1 Funktionaalinen paradigma

Eräs suurimpiin muutoksiin kuuluva asia oli ohjelmointikielen vaihto. Aikaisemmasta oliopohjaisesta Javasta vaihdettiin funktionaaliseen Clojure-kieleen. Projektin reunaehtona oli valita ohjelmointikieli, jota ajetaan JVM:ssä. Muutoksella pyrittiin hyödyntämään kielen parempaa soveltuvuutta datan muokkaukseen ja käsittelyyn. Clojure valikoitui käytettäväksi funktionaaliseksi kieleksi, koska osalla kehittäjistä oli siitä hyviä aikaisempia kokemuksia. Puoltavana asiana oli myös Clojure-koodin ilmaisuvoimaisuus sekä tiiviys verrattuna Javaan. Funktionaalisen kielen takia alkuperäisen järjestelmän luokkahierarkiat väistyivät funktioiden tieltä. Ohjelmointikielen vaihdon ei arvioitu aiheuttavan suurta lisätyötä lukuun ottamatta uuden paradigman omaksumista. Aikaisempaa ohjelmakoodia ei olisi voinut hyödyntää paljoakaan, koska arkkitehtuuriin ja tietomalleihin kohdistui niin suuria muutoksia.

Web-sovelluksessa selaimessa ajettava ohjelma on usein erotettu palvelinsovelluksesta. Nämä kommunikoivat keskenään rajapinnan avulla. Koodin uudelleenkäytön kannalta on hyödyllistä, jos samaa ohjelmakoodia voidaan hyödyntää molemmissa päissä. Selaimessa suoritettava ohjelma on pohjimmiltaan JavaScript-kielellä tehty. Palvelinpäässä

toteutuskielenä on jokin toinen ohjelmointikieli. Clojure tarjoaa kätevän keinon tähän asiaan. Ohjelmakoodi voidaan jakaa puhtaaseen Clojure-kieleen (tiedostopäätteenä .clj), niin sanottuun yhteiseen osaan (.cljc) ja web-käyttöliittymän ClojureScriptiin (.cljs). Yhteinen osa on tarkoitettu sekä JVM:ssä suoritettavaksi että JavaScriptiksi käännettäväksi [13]. Yhteisen koodin käyttö mahdollistaa esimerkiksi syötettävien tietojen validoinnin palvelimessa ja dynaamisesti käyttöliittymässä ilman uuden koodin kirjoitusta samaan toteutukseen pohjautuen. Selkeänä hyötynä on koodin uudelleenkäytettävyys ja ylläpidettävyys pienemmän koodimäärän ansiosta. Kuvassa 5 näkyy selkeämmin, millä tasoilla käytetään kunkin tyyppistä Clojure-kieltä.

Web-toimintaympäristönä on luonteeltaan tapahtumiin reagoimista. Palvelinohjelma tarjoaa HTTP-rajapinnan ja reagoi käyttäjien pyyntöihin. Toisaalta sama malli toistuu loppukäyttäjien käyttöliittymässä. Selain näyttää staattista sisältöä, kunnes käyttäjä tekee jonkin toimenpiteen. Tähän tilanteeseen funktionaalinen reaktiivinen ohjelmointi sopii erinomaisesti. Palvelin palvelee monia käyttäjiä samanaikaisesti. Funktionaalinen paradigma ja muuttumattomuus sopivat tällaiseen samanaikaiseen toimintaan. Web-käyttöliittymä korvattiin ClojureScriptillä tehdyksi Reactiin pohjautuvaksi dynaamiseksi SPA-toteutukseksi. Muutoksella liikenteen määrää ja palvelimen kuormitusta saatiin vähennettyä alkuperäisestä Wicket-AJAX -yhdistelmästä. Funktionaalinen reaktiivinen ajatusmalli ulottui web-käyttöliittymän lisäksi myös palvelimen ja käyttöliittymän väliseen molemmin suuntaiseen kommunikointiin WebSocket-pohjaisen liikenteen avulla.

6.3.2 Datamalli

Sovelluksen datamallia ajateltiin uudelta kannalta. Tätä yksinkertaistettiin soveltumaan paremmin käyttötarkoitukseen. Vanhassa järjestelmässä Java-koodin luokkahierarkiat heijastuivat tietokannan rakenteisiin asti, mikä vaikeutti rakenteen ymmärtämistä. Uudistuksen yhteydessä datamalli suunniteltiin uudelleen. Uudelleensuunnittelussa huomioitiin operatiivisen toiminnan lisäksi myös muut tarpeet kuten raportointi. Datamallin muutokseen myötävaikutti toisen tietokantamoottorin käyttöönotto. Aikaisemmasta Ingres-tietokannasta vaihdettiin huomattavasti nykyaikaisempaan PostgreSQL-tietokantaan, minkä ansiosta esimerkiksi dynaamiset tietorakenteet olivat mahdollisia.

Suorituskykyongelmat johtuivat sekä tietokantarakenteesta että datan käsittelystä. Tietokantarakenne oli suunniteltu sisältämään vain muutoksia, ei nykytilaa. Sovelluksen ehdottomasti suurin käyttötarkoitus oli kuitenkin nykytilan näyttäminen. Aikaisemmin käytettiin niin sanottua event sourcing -mallia, jossa nykytila muodostetaan summaamalla kaikki muutokset yhteen. Tämä on toimiva ja tehokas malli, mikäli laskennan voi hoitaa

tietokannassa. Vanhan sovelluksen tapauksessa laskemiseen tarvittiin kuitenkin raakadatan lisäksi tuhansia rivejä ohjelmakoodia kontekstin selvittämiseksi ja datan tulkitsemiseksi. Tämä hidasti suorituskykyä oleellisesti, koska datan käsittely tehtiin pääasiassa sovelluksen puolella ja jokainen tallennettu muutos laskettiin ohjelmakoodissa eikä siinä hyödynnetty tietokantamoottorin palveluita. Datan tallennusmuoto oli sellainen, että tietokantamoottorin hyödyntäminen laskennassa oli hankalaa. Uudistuksen myötä tietokantaan varastoidaan nykytila, jota päivitetään datan tallennuksessa, ja yksinkertainen laskenta tehdään mahdollisimman paljon tietokannan puolella.

Sovellukseen tehtyjen muutosten lisäksi tietokantaan tallennetun datan rakennetta muutettiin. Vaatimuksena oli myös olemassa olevan alkuperäisen järjestelmän datan säilyttäminen. Yhdessä ne aiheuttivat tarpeen datan migraatiolle vanhasta rakenteesta uuteen datamalliin. Migraation tekeminen aiheutti ennakkoarvioon nähden eniten haasteita. Datan muuttaminen muodosta toiseen edellytti vahvaa tuntemusta sekä vanhasta että uudesta järjestelmästä. Osa uuteen järjestelmään tehdyistä rajoitteista ei pätenyt vanhassa järjestelmässä tai vanhassa oli ollut jokin virhe, jonka seurauksena järjestelmään oli tallentunut vääränlaista dataa. Dataa ei voinut kuitenkaan enää jälkikäteen muokata oikeaksi vaan se oli tallennettava samanlaisena, jotta järjestelmät toimisivat mahdollisimman samanlaisesti. Datamigraatioon kului huomattavan suuri osa kehitysajasta ja jälkikäteen ajatellen tämän olisi voinut erottaa omaksi projektikseen.

7. MITTAUSTULOKSET JA NIIDEN ANALYSOINTI

Luvussa kuvataan mittausten lähtökohta ja oletetut tulokset. Varsinaiset mitatut tulokset esitetään jaoteltuna ohjelmiston laatuun ja haastatteluihin. Ohjelmistolaadun mittarit ovat enimmäkseen kvantitatiivisia, kun puolestaan haastatteluissa selkeä painopiste on laadullisessa analyysissa. Tuloksista esitetään raakadata, jonka perusteella arvioidaan onnistumista. Lopuksi tehdään kokonaisarvio uudistuksesta.

7.1 Mittausten lähtökohta

Ohjelmiston uudistamista arvioidaan haastattelulla ja arvioimalla koodin laatua. Onnistumisen arvioinnissa selkeä pääpaino on haastattelulla, koska osa koodin laatuun liittyvistä mittareista perustuu suuremman aikavälin seurantaan tuotantokäytön aloittamisen jälkeen. Tällainen seuranta on mahdollista tehdä vasta myöhemmin.

Uuden sovelluksen koodimäärää verrataan vanhaan. Koodimäärän pitäisi vähentyä jo pelkästään ohjelmointikielen vaihdon seurauksena, mutta eniten saavutettavaa on teknisen rakenteen ja arkkitehtuurin yksinkertaistumisesta ja tätä kautta paremmasta ylläpidettävyydestä. Koodin määrän pienenemisellä tavoitellaan parempaa ylläpidettävyyttä ja ohjelman ymmärrettävyyttä koodin perusteella.

Sovelluksen teknisen rakenteen selkeyttä arvioidaan valittuun toimenpiteeseen liittyvien funktiokutsujen määrän avulla. Arviointitapa ei ole täysin aukoton, koska yksinkertaisempien tietorakenteiden lopputuloksena data voi päätyä toisten toimintojen kannalta hankalammin hyödynnettävään muotoon, minkä selvittäminen on todella vaikeaa. Tämän arvioinnin perusteella voidaan kuitenkin tehokkaasti arvioida uuden vastaavan toiminnon tekemistä sekä muutoksien tekemistä olemassa oleviin toimintoihin.

Tietokannan rakenteen ymmärtäminen liittyy esimerkiksi raportointiin ja virheellisen datan korjaamiseen jälkikäteen. Vanhan ja uudistetun järjestelmän tietokantojen rakenteita verrataan keskenään. Lähtökohtaisesti voidaan ajatella, että yksinkertaisempi tietokantarakenne on parempi ymmärrettävyyden kannalta. Tässä on tosin kääntöpuolena liian yleiskäyttöinen tietokantarakenne esimerkiksi dynaamisia tietotyyppejä käytettäessä, mikä vaikeuttaa oleellisen datan hahmottamista.

Ohjelmien testien koodikattavuutta ei voi verrata keskenään, koska vanhassa järjestelmässä oli vain muutamia testejä. Eräänä uudistuksen lähtökohtana oli helpottaa jatkokehitystä laatimalla yksikkö-, integraatio- tai järjestelmätason testit kaikista ohjelman ominaisuuksista. Kattavien testien avulla voidaan vähentää regressiovirheiden syntyä eli

vältetään virheiden syntymistä ennestään toimiviin asioihin uusien muutosten tekemisen yhteydessä. Testien lisäämisellä helpotetaan myöhempää kehitystä ja ylläpitoa.

7.2 Ohjelmiston metriikat

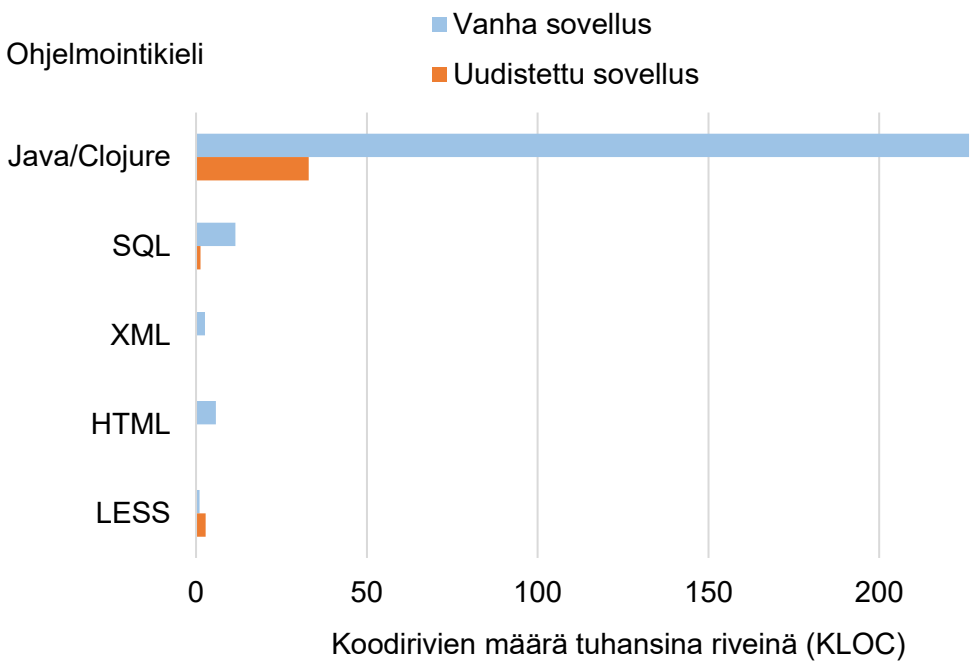
Ohjelmistojen metriikoilla tarkoitetaan ohjelmistojen ominaisuuksien mittaamista. Mittaamalla saadaan tietoa ohjelmiston kompleksisuudesta ja laadusta. [34] Ohjelmiston laatua arvioidaan koodirivien ja tiedostojen määrällä, tietokantarakenteen perusteella ja valitun toiminnon monimutkaisuuden perusteella. Mittarit ovat keskittyneet nykytilan tutkimiseen eikä pidemmän aikavälin tutkimusta ole mahdollista tehdä. Pidemmältä aikaväliltä tilastotietoa voisi kerätä esimerkiksi virheiden määrästä, jonka voi suhteuttaa koodirivien määrään.

7.2.1 Ohjelmakoodi

Uudistuksen eräänä lähtökohtana oli ylläpidon helpottaminen. Tähän pyrittiin muun muassa koodirivien määrän vähentämisellä. Uudistetun ohjelman koodirivien määrä ohjelmointikielittäin on esitetty taulukossa 6. Taulukosta on jätetty pois sellaiset tiedostot, joiden yhteenlaskettu koodimäärä on alle tuhat riviä.

Taulukko 6 *Uudistetun ohjelman koodirivien määrä ohjelmointikielittäin eroteltuna*

Ohjelmointikieli	Koodirivien määrä tuhansina riveinä (KLOC)
Clojure	20,4
ClojureScript	8,1
Clojure (.cljc)	4,5
LESS	2,8
SQL	1,3



Kuva 6 *Vanhan ja uudistetun sovelluksen koodirivien määrä*

Kuvassa 6 havainnollistetaan muutosta vanhan ja uudistetun sovelluksen välillä. Kuvassa Java- ja Clojure-koodin rivimääriä vertaillaan keskenään. Clojuren alle on yhdistetty Clojure, ClojureScript ja yhteinen Clojure-koodi.

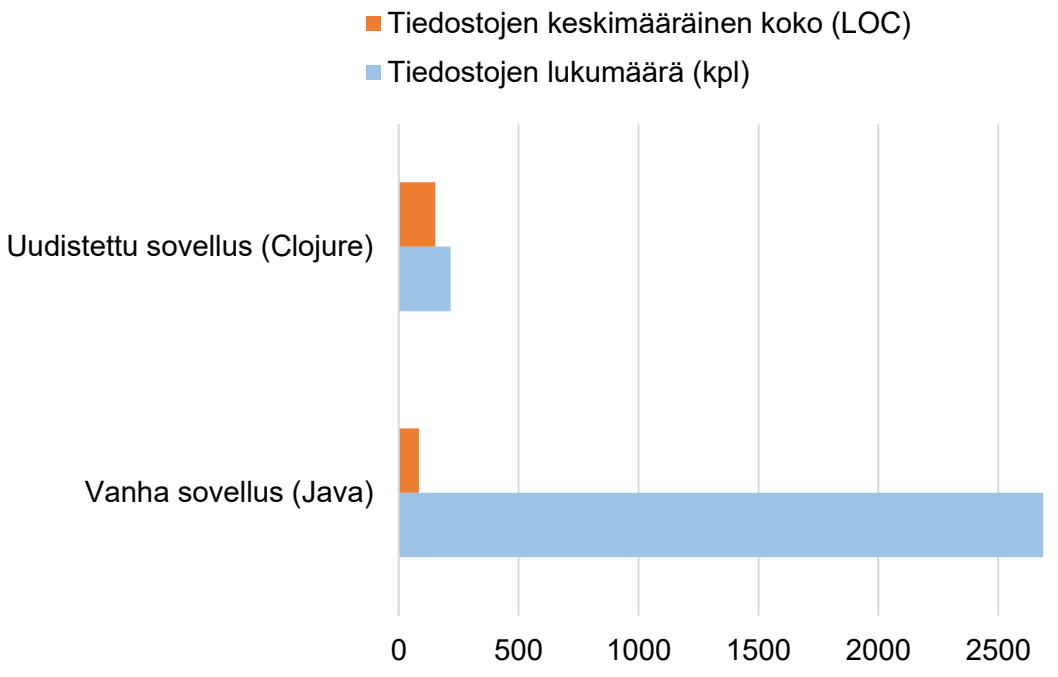
Uudistettu sovellus on lähes kaikin ohjelmointikielin tarkasteltuna reilusti pienempi kuin vanha sovellus. Uudistuksen jälkeen ohjelman koodipohja on noin 85 % pienempi kuin ennen uudistusta. Ehdottomasti suurin vähennys tulee Javan ja Clojuren välillä. Muuten vähennys on ollut maltillisempaa. Koodirivien määrä on kasvanut uudistuksen myötä LESS-koodissa lähes kolminkertaiseksi. Tämän selittää koko järjestelmän yhteisten tyylitiedostojen hyödyntäminen vanhassa järjestelmässä. Tätä ei ole tehty uudistetussa sovelluksessa.

Koodimäärä ei ole täysin vertailukelpoinen, koska uudistetussa ohjelmassa ei ole kaikkia samoja ominaisuuksia kuin vanhassa ja osa toteutuksesta vielä puuttuu. Nykyinen toiminnallisuus kattaa noin 75 % valmiista sovelluksesta. Tämä asia huomioiden uudistuksen jälkeinen koodirivien määrä on silti huomattavasti pienempi, noin viidesosa alkuperäisestä. Ulkoisten kirjastojen määrässä ei ollut merkittävästi eroa. Kirjastoja on molemmissa toteutuksissa kymmeniä.

Koodirivien määrän pienenemiselle ei ole yhtä yksittäistä syytä. Ensimmäisenä asiana pienenemiseen on vaikuttanut vahvasti ohjelmistoarkkitehtuuriin tehdyt yksinkertaistuk-

set. Toisena asiana on Clojuren käyttöönotto, jonka seurauksena rivimäärä on pienentynyt. Koodimäärän pieneneminen on havaittu myös muissa tutkimuksissa. Erään tutkimuksen [2] mukaan Clojure-koodin pituuden on arvioitu olevan noin neljäsosa vastaavan Java-koodin pituudesta. Toisessa tutkimuksessa [16] on vertailtu rivimäärää LISP:n ja Javan välillä. Tuloksena on havaittu LISP-ohjelmien olevan noin puolet pienempiä. Pieneneminen on johtunut implisiittisestä tyypityksestä ja kielen korkeammista abstraktiotasoista. LISP on syntaksiltaan ja periaatteiltaan hyvin samankaltainen Clojuren kanssa, joten sitä voi käyttää vertailukohtana.

Ohjelmakoodin tiedostojen määrä kertoo Java-toteutuksessa luokkien määrästä. Clojure-toteutuksessa tiedostot kertovat nimiavaruuksien määrästä. Vanhan ja uudistetun sovelluksen tiedostojen määrät on esitetty kuvassa 7. Kuvassa on laskettu vain ohjelmakoodia sisältävät tiedostot eli vanhassa toteutuksessa .java-päätteiset tiedostot ja Clojure-toteutuksessa .clj-, .cljs- ja .cljc -tiedostot. Näiden lisäksi toteutukset sisältävät muun muassa HTML-, JavaScript- ja CSS-tiedostoja sekä konfiguraatiotiedostoja, mutta näiden yhteenlaskettu määrä on minimaalinen kokonaisuuden kannalta.



Kuva 7 *Sovelluksien tiedostojen lukumäärä sekä niiden keskimääräinen koko*

Tiedostojen lukumäärä on uudistetussa sovelluksessa paljon pienempi, mutta tiedostojen keskimääräinen pituus on kasvanut. Tiedostojen koon kasvu johtuu muun muassa

siitä, että Clojuressa tieto välitetään ohjelmakoodin sisällä tyypillisesti map-tietorakenteen avulla. Java-koodissa käytetään tyypillisesti DTO:ita (tiedonvälitysolio, *data transfer object*) tiedon välittämiseen ja JPA-entiteettejä tietokannan rakenteen kuvaamiseen. Näistä molemmista muodostuu kooltaan pieniä luokkia, jotka eivät sisällä juurikaan toiminnallisuutta. Myös Java-koodissa olevat lukuisat rajapinnat lisäävät tiedostojen määrä. Nämä eivät sisällä toteutusta, joten keskimääräinen tiedostojen koko pienenee.

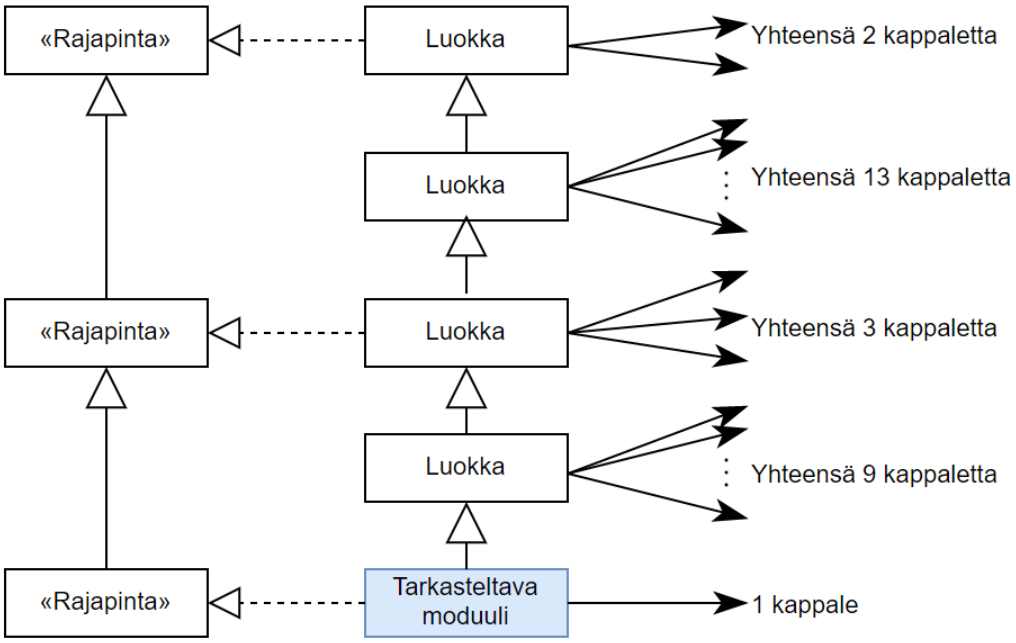
7.2.2 Tietokanta

Datamallin uudelleen suunnittelu vaikutti tietokantarakenteeseen. Vanhaan sovellukseen liittyi 106 tietokantataulua. Näkymiä ei ollut. Uudistetussa sovelluksessa on 18 tietokantataulua ja kaksi näkymää. Tietokantataulujen määrän pienenemisellä näyttää olevan vahva korrelaatio koodirivien määrän pienenemiseen, sillä prosentuaalisesti tarkasteltuna pieneneminen on ollut aivan yhtä suurta. Tämä on loogista, koska Java-koodissa jokaista tietokantataulua vastaa yksi luokka, jolloin tietorakenteita yksinkertaistamalla myös luokkien määrä pienenee. Tosin ohjelmakoodissa luokkamäärän pieneneminen kohdistuu monikerrosarkkitehtuurissa vain yhdelle tasolle.

Taulumäärän pieneneminen johtui tehtävien erityyppisten toimenpiteiden yleistyksellä. Uudistuksen myötä nämä tallennetaan samaan tietokantatauluun. Vanhassa sovelluksessa jokaiselle eri toimenpidetyypille oli yhteisten tietokantataulujen lisäksi vielä toimenpidetyyppikohtainen taulu. Vanhassa sovelluksessa jokaiselle tiedolle on tarkkaan määritelty oma kenttänsä. Uudistetussa sovelluksessa on staattisten tietotyyppien lisäksi hyödynnetty dynaamista JSON-tietotyyppiä, jonka sisäistä rakennetta voi helposti laajentaa. JSON-muotoon on yksinkertaista tallentaa eri tyyppisiin toimenpiteisiin liittyvät omat erikoisuutensa.

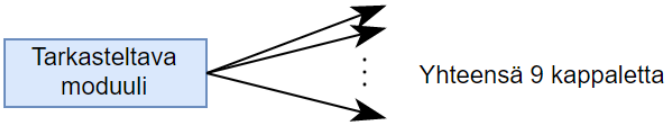
7.2.3 Ohjelman rakenne

Ohjelman sisäistä rakennetta voi tarkastella luokkakaavion avulla. Luokkakaavio kuvaa jonkin tietyn osan tai koko järjestelmän luokat ja niiden väliset keskinäiset suhteet. Vanhan järjestelmän kuvaus on esitetty kuvassa 8. Tarkasteltavaksi luokaksi on valittu erään tyypillisen prosessin sisältävä luokka. Varsinainen tarkasteltava luokka on korostettu kuvaan sinisellä. Luokka periytyy yhteensä neljästä yläluokasta ja toteuttaa rajapintoja kolmessa eri tasossa. Jokaisella yläluokalla on omia riippuvuuksiaan. Näiden injektiopisteiden määrä on merkitty oikeaan laitaan. Yhteensä injektiopisteitä on 28 kappaletta.



Kuva 8 *Vanhan järjestelmän luokkakaavio ja kutsuriippuvuudet*

Vanhaa toteutusta vastaavaa luokkakaaviota ei voi suoraan muodostaa uudistetusta ohjelmasta, koska uusi ei sisällä luokkia. Luokkien tilalla voidaan käyttää nimiavaruuksia, joiden avulla kapselointia voidaan tehdä. Uudistetun sovelluksen vastaavaan prosessitoiminnallisuuden sisältävä toteutus on esitetty kuvassa 9. Injektiopisteiden tilalla on laskettu riippuvuudet sivuvaikutuksia aiheuttaviin toisiin nimiavaruuksiin.



Kuva 9 *Uudistetun järjestelmän kutsuriippuvuudet*

Kuvien 8 ja 9 perusteella tarkasteltuna uudistettu sovellus on rakenteeltaan paljon yksinkertaisempi, mikä auttaa ohjelman ymmärtämisessä ja ylläpidettävyydessä. Toisaalta kuvat esittävät vain erään tarkastelupisteen rakenteen, joten niiden perusteella ei voi kiistattomasti tehdä johtopäätöksiä koko järjestelmästä, vaikka tarkasteltava kohta onkin valittu mahdollisimman tyypillisen osan kattavasta toiminnallisuudesta. Ohjelman rakenteen havainnekuvista ei selviä yksittäisten luokkien tai nimiavaruuksien koot. Vanhan toteutuksen luokkahierarkian voisi typistää sisältymään yhteen luokkaan, jolloin hierarkia

olisi matalampi, mutta yksittäinen luokka kooltaan huomattavasti suurempi. Uudistetussa sovelluksessa käsiteltävä nimiavaruus on koodirivein mitattuna samaa suuruusluokkaa kuin vanhan sovelluksen alin taso. Nämä kaksi keskenään yhtä suurta osaa on korostettu kuvissa sinisellä värillä. Nimiavaruuden paisuttaminen ei ole ollut ratkaisuna, vaan rakenne on myös oikeasti yksinkertaisempi. Tätä tukee myös paljon pienempi lähteviä kutsuja sisältävien riippuvuuksien määrä.

7.3 Havainnot

Uudistuksen tekemisen aikana tehtiin havaintoja erityisesti funktionaalisesta paradigmasta ja Clojuresta kielenä. Uudistuksessa oli mukana ensin neljä ohjelmistokehittäjää, joiden määrä kasvoi projektin aikana kuuteen. Näistä kahdella oli vankka Clojure-osaaminen ennestään. Kaikilla kehittäjillä oli reilusti kokemusta Java- ja oliopohjaisista kielistä.

Clojuren opetteluun kului muutamia kuukausia ennen kuin se alkoi sujua vaivattomasti. Tähän vaikutti funktionaaliseen ohjelmointiparadigmaan sisälle pääseminen ja vieraampi syntaksi, jossa funktio on ensimmäisenä. Clojure sisältää paljon valmiita funktioita, joiden olemassaolosta ei kehittäjillä kokemuksen puutteen vuoksi ollut tietoa. Tämä aiheutti tarpeettoman monimutkaista koodia, jonka toiminnallisuuden tekemiseen olisi ollut helpompiakin tapoja. Funktionaaliseen ohjelmointiparadigman ja Clojuren oppimiseen auttoi katselmointikäytäntö, joissa Clojure-osaajat kommentoivat koodia ja antoivat vinkkejä kokemattomille kehittäjille.

Dynaaminen tyypitys aiheutti hankaluuksia. Funktioita käytettäessä ei aina ollut selvää, mitä dataa pitäisi välittää tai missä muodossa sen pitäisi olla. Argumenttien nimeämisessä voidaan toki käyttää erilaisia konventioita, esimerkiksi unkarilaista notaatioita, jossa muuttujan nimi paljastaa myös sen tyypin, mutta se ei sovellu kovinkaan monimutkaisiin dynaamisiin tietorakenteisiin. Yhtenä vaihtoehtona olisi käyttää sellaista ohjelmointikieltä, jossa tyypit olisivat staattisia ja ne pitäisi kuvata ohjelmakoodia kirjoittaessa. Ohjelman virheiden jäljitys (*debugging*) koettiin vaikeammaksi esimerkiksi tietorakenteiden laiskuuden takia. Virhe saattoi olla lähtöisin aivan muualta kuin ensin alkuun näyttikään. Lisäksi ohjelman suoritus halutulla tavalla virheenjäljitystilassa ei ollut kovinkaan suoraviivaista.

Havaintojen positiiviset seikat liittyivät osittain samoihin asioihin kuin negatiiviset. Positiiviseksi asiaksi koettiin datan muuttumattomuus. Kehittäjät voivat olla varmoja, että mikään funktio ei voi muuttaa saamiensa argumenttien arvoja. Mikäli datan haluttaisiin

muuttuvan, pitää sijoitus tehdä funktion kutsujan päässä. Ohjelman kehittämisessä hyödynnettiin Clojuren tarjoamaa REPL-työkalua. Tämän avulla sovellus voi olla päällä koko ajan. Muutokset voidaan ladata ohjelman suoritukseen ajonaikaisesti käynnistämättä sovellusta uudelleen. Perinteisesti Java-ohjelmissa muutoksien jälkeen tarvitsee kääntää koodit ja käynnistää palvelin uudelleen. REPL:iä voitiin hyödyntää myös virheiden jäljitykseen tarkastelemalla nimiavaruuksien sisältöä interaktiivisesti. Tietoa voitiin tallentaa muuttujiin, joita voitiin tarkastella REPL:n avulla.

Kaiken kaikkiaan Clojuresta ja funktionaalisesta ajatusmallista jäi positiivinen yleiskuva. Kehittäjät kokivat olevansa tuottavampia ja pitivät erityisesti Clojuren ilmaisuvoimasta. Clojureen perehtyminen ja opettelu edellyttävät kuitenkin aikaa.

Ohjelman sisäinen rakenne koettiin yksinkertaiseksi ja helpommaksi ymmärtää. Ohjelman rakenteen ymmärtämisestä kertoo se seikka, että uusien toimintojen tekeminen aloitettiin ennakkoluulottomasti ja niissä päästiin nopeasti eteenpäin. Arkkitehtuuritasolla oli luotu raamit, joiden puitteissa uusia toimintoja oli suoraviivaista luoda, ja joka ei rajoittanut tekemistä liikaa.

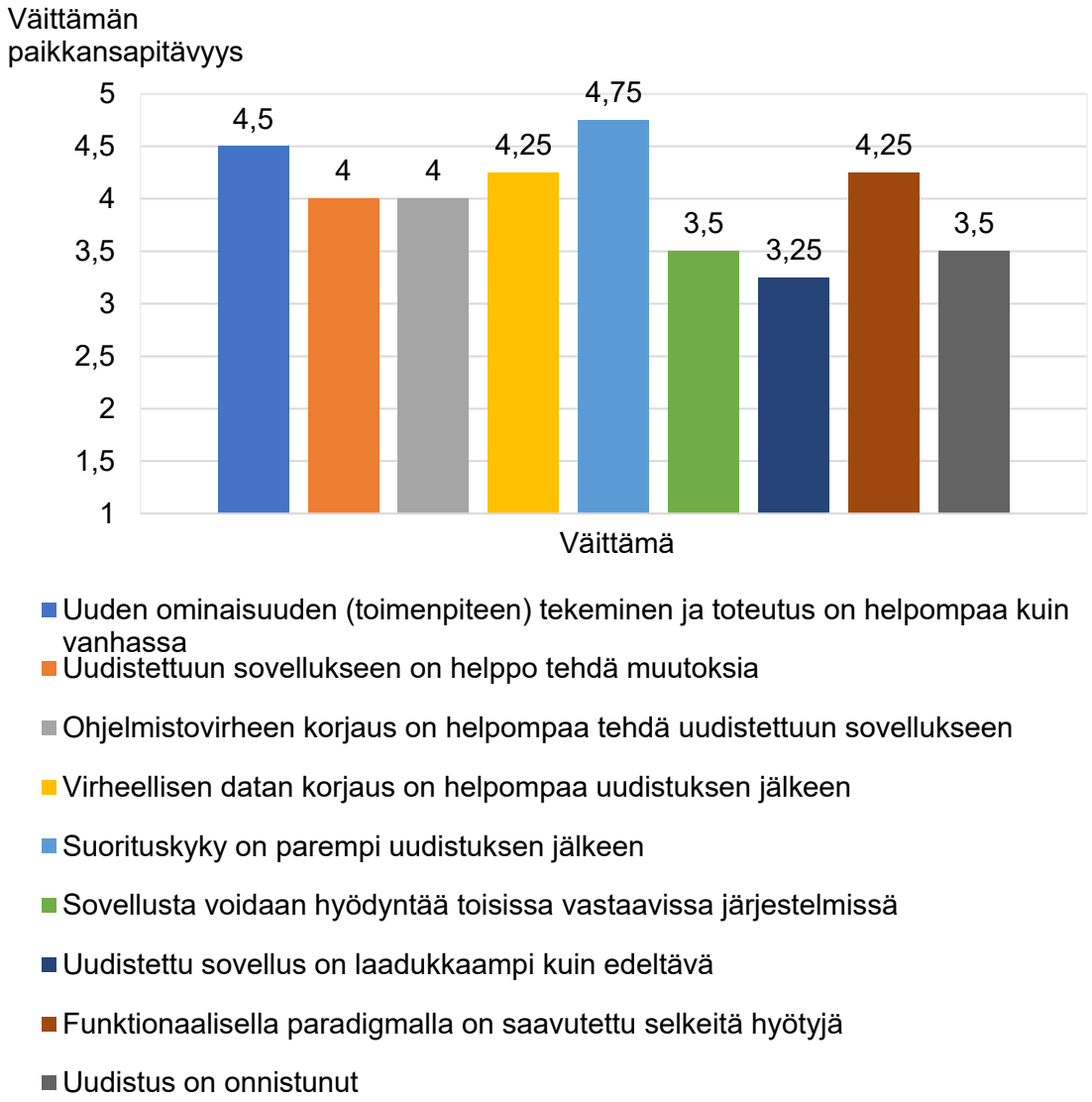
7.4 Kysely

Kysely koostui yhdeksästä väittämästä, joiden paikkansapitävyyttä arvioitiin asteikolla yhdestä viiteen, joista viisi tarkoittaa, että on väittämän kanssa samaa mieltä, ja yksi, että on täysin eri mieltä. Asteikon keskimmäinen arvo, kolme, tarkoittaa, että ei ole samaa eikä eri mieltä väittämän kanssa. Kaiken kaikkiaan kyselyyn osallistujia oli neljä henkilöä. Kyselyn päällimmäisin tehtävä oli pohjustaa myöhemmin käytäviä haastatteluja. Tehdyn kyselyn tulokset on esitetty kuvassa 10.

Tuloksissa kahtena eniten onnistuneena asiana esiin nousi uuden ominaisuuden lisäämisen helppous sekä suorituskyvyn parannus. Ensimmäisenä mainittu uuden ominaisuuden lisääminen edellyttää toimialueen tuntemuksen lisäksi ohjelman teknistä ymmärtämistä. Ohjelman ymmärtämiseen auttaa yksinkertaisempi tekninen rakenne. Suorituskyvyn parantuminen on jokapäiväisessä käytössä havaittu ja suurimmat tähän vaikuttava asiat ovat datamallin uudelleensuunnittelu ja tietokannan tehokas käyttö.

Kaksi vähiten onnistunutta asiaa olivat laadukkuus suhteessa vanhaan järjestelmään ja uudistetun sovelluksen hyödyntäminen muissa vastaavissa projekteissa. Uudistetun sovelluksen laadukkuus on samaa luokkaa kuin vanhan sovelluksen. Kyselyn perusteella uudistetun ohjelman hyödyntäminen muissa projekteissa on mahdollista, mutta ei todennäköistä.

Eräs mielenkiintoinen seikka on, että uudistuksen onnistumisen arviointiin vastattiin hyvin varovaisesti. Tulos on lähellä neutraalia, vaikka osa-alueittain tarkasteltuna uudistuksella oli selkeitä hyötyjä. Osaltaan tähän vaikuttaa, että uudistettua sovellusta ei vielä ole otettu tuotantokäyttöön. Kaiken kaikkiaan kyselyn perusteella voidaan todeta, että uudistuksessa mentiin oikeaan suuntaan, koska mikään tarkasteltu osa-alue ei ollut huonompi kuin vanhassa sovelluksessa. Kyselyn vastauksen perustuivat senhetkisiin arvioihin. Uudistuksen onnistumisen arvioinnin kannalta tilannetta pitää seurata ja parempi arvio saadaan vasta pidemmän ajan kuluessa.



Kuva 10 *Kyselyn tulosten keskiarvot*

7.5 Haastattelu

Haastatteluun osallistuivat samat henkilöt, jotka osallistuivat kyselyyn. Ohjelmistokehittäjien haastatteluissa tarkasteltiin enemmän uudistamista tekniseltä kannata. Tuoteomistajan haastattelussa huomio kiinnittyi toiminnallisuuksiin ja projektinhallintaan. Haastattelun tavoitteena oli poimia perusteluita kyselyssä annettuihin vastauksiin. Lisäksi haastattelu antoi mahdollisuuden kertoa sellaisista asioista, joita kyselyssä ei ollut huomioitu.

Haastatteluissa arvioitiin, millä tapaa uusi sovellus on parempi kuin vanha. Useampi haastateltava vastasi, että uudistuksen myötä teknologiat ovat modernimpia, sovelluksen tekninen rakenne on helpompi ymmärtää ja hahmottaa. Lisäksi luettavuus on parempi. Sovellus on joustavampi ja tietorakenteet dynaamisempia. Uutta toiminnallisuutta voidaan tehdä ilman tietokantamuutoksia, mikä ei onnistunut vanhassa sovelluksessa. Eri osa-alueet on erotettu sekä teknisesti että toiminnallisesti paremmin toisistaan. Toiminnallisuuden kannalta käyttökokemus on parantunut. Käyttöliittymä on intuitiivisempi ja vasteajat pienempiä.

Vanha sovelluksen vahvuuksiksi kerrottiin hyvät ja johdonmukaiset ohjelmointikäytännöt, sisäisen rakenteen loogisuus ja yleisten suunnitteluperiaatteiden käyttö. Vanha sovellus toimi luotettavasti ja vastasi toiminnallisuudeltaan alkuperäisiä suunnitelmia. Haastatteluissa kävi ilmi, että vanhan sovelluksen rakenne ja laatu on heikentynyt ajan myötä. Lehmanin seitsemäs laki laadun heikkenemisestä on siis konkretisoitunut.

Javan korvaaminen funktionaalisella Clojurella nähtiin ohjelmistokehittäjien kannalta innostavana ja mielenkiintoisena vaihteluna. Ohjelmakoodista on tullut selkeämpää ja sen ymmärtäminen on helpompaa. Funktioiden puhtaus ja datan muuttumattomuus helpottavat ymmärtämistä ja vähentävät virheitä. Funktionaalinen kieli koettiin sopivan hyvin ohjelmallista datakäsittelyä sisältäviin ohjelmiin. Clojuren hyviksi puoliksi mainittiin REPL, jonka ansiosta kehittäminen on nopeaa. Tehdyt koodimuutokset on mahdollista ottaa käyttöön käynnistämättä koko palvelinta uudelleen. Heikkoudeksi nähtiin tietorakenteiden huonompi läpinäkyvyys. Javassa jokaisen luokan rakenteen voi tarkistaa luokan määrittelystä. Clojuressa tietorakenteiden selvittäminen koettiin hankalaksi niiden dynaamisuuden takia. Tätä asiaa helpottamaan on kehitetty spec-kirjasto [3], jonka avulla voidaan määritellä tietorakenteita.

Suurimmalla osalla kehittäjistä oli vankka kokemus olio-ohjelmoinnista ennestään ja funktionaalisen paradigman omaksuminen vei aikaa. Toisena aluksi hankalana asiana nähtiin Clojuren syntaksi. Kaiken kaikkiaan uuden paradigman ja ohjelmointikielen omaksumisen arvioitiin vievän noin puoli vuotta. Tämän jälkeen tuottavuus ei vielä ole

huipussaan, vaan paradigma vaatii edelleen opettelua. Puolen vuoden perehtymisen jälkeen kehittäjät kokivat kuitenkin pystyvänsä tekemään uusia ominaisuuksia tehokkaasti.

Osakokonaisuus irrotettiin muusta sovelluksesta uudistamisen yhteydessä. Tällä oli sekä hyviä että huonoja puolia. Hyväksi asiaksi nähtiin idea loogisen kokonaisuuden irrottamisesta erilliseksi kokonaisuudeksi, joka toimii mahdollisimman itsenäisesti. Sen kehittäminen on vaivattomampaa, koska muuta järjestelmäkokonaisuutta ei tarvitse käynnistää tai käyttää ja kehitysympäristö on tällä tavoin huomattavasti kevyempi. Huonona asiana nähtiin datan eheyden varmistamisen monimutkaistuminen. Aikaisemmin tapahtumat suoritettiin yhden transaktion sisällä, jolloin virheistä palautuminen oli triviaalia. Uudistetussa sovelluksessa tämä pitää erikseen huomioida, koska eri järjestelmien toiminta on hajautettua.

Datamallin uudelleensuunnittelun myötä suorituskyky on parantunut, millä on suora vaikutus ohjelman käyttöön ja loppukäyttäjille. Ohjelmistokehittäjien kannalta suorituskyvyn parantumisella on myös hyviä puolia kuten esimerkiksi testien ajaminen on nopeampaa. Datan tietokantarakenne on yksinkertaisempi, minkä ansiosta virheellisen datan korjaus on helpompaa kuin vanhassa sovelluksessa.

Uudistetun sovelluksen koettiin palvelevan niin ominaispiirteistä toimialaa ja toiminnallisuutta, että sitä ei suoraan uskottu käytettävän toisessa projektissa. Ohjelmistokehittäjät kuitenkin mainitsivat sen sisältävän hyviä konsepteja, joita voitaisiin käyttää uudelleen esimerkiksi toimintaprosesseja tai monimutkaisia web-lomakkeita sisältävissä sovelluksissa.

Haastateltavia pyydettiin arvioimaan uudistuksen tarkastelemista projektina. Hyvin menneitä asioita olivat ennakkoluuloton ja rohkea lähestymistapa vanhan sovelluksen tietomalleihin ja käsitteisiin. Uskallettiin tehdä suuriakin muutoksia. Positiivista oli myös loppukäyttäjien aktiivinen osallistuminen uudistukseen. Teknologioista siirryttiin modernimpiin lähes kaikilla eri tasoilla. Tietokantamoottori vaihdettiin uudempaan, vanhasta Java-versiosta vaihdettiin Clojureen ja käyttöliittymässä siirryttiin nykyaikaiseen JavaScript-pohjaiseen toteutukseen.

Funktionaalisesta paradigmasta koettiin olevan hyötyä esimerkkitapauksessa. Ohjelmointikielen valinnasta sen sijaan ei ollut yhtä suurta yksimielisyyttä. Muiksi vaihtoehdoiksi mainittiin muun muassa Scala, Javan uudemmat versiot ja Kotlin. Scalan vahvuuksiksi nähtiin käännösaikaiset tyyppitarkistukset, jolloin ajonaikaisia virheitä on vähemmän. Uudemmilla Javan versioilla koodin määrän ei arvioitu vähentyvän juurikaan alkuperäisestä. Streamit ja lambda-funktiot nähtiin positiivisena asiana, mutta virheiden et-

sintä arvioitiin haastavammaksi. Kotlinin arvioitiin olevan vähemmän soveltuva datankäsittelyyn, vaikka se sisältääkin puhtaaseen Javaan verrattuna parannusta. Clojuren valintaa puolsi ohjelmakoodin tiivis esitysmalli sekä hyvät kokemukset Clojurea ennestään käyttäneillä kehittäjillä.

Uudistusprojektissa parannettavaa löytyi esiselvityksestä ja suunnittelusta. Projektin alkuvaiheessa olisi pitänyt perehtyä vielä enemmän toimialaan ja sen erityispiirteisiin sekä sovelluksen kontekstiin. Selkeästi suurimpana yksittäisenä asiana haastatteluissa nousi esille datamigraatio vanhasta järjestelmästä uuteen. Se vei huomattavasti enemmän aikaa kuin alun perin ajateltiin. Migraatioiden tekeminen edellytti kattavaa uudistetun sovelluksen ja vanhan järjestelmän datarakenteiden ja sovelluslogiikan tuntemusta.

7.6 Uudistuksen onnistumisen arviointi

Puhtaasti teknisiin metriikoihin perustuen uudistus oli onnistunut. Datamalli, tietokantarakenteet ja ohjelmakoodin sisäinen rakenne yksinkertaistuivat. Tämä oli osaltaan odotettavaa, koska vanhan ja uuden sovelluksen ominaisuudet eivät ole täysin yhteneväiset. Erityisen huomattavaa oli ohjelman koodirivien ja tiedostojen määrän pieneneminen.

Teknisellä yksinkertaistumisella on suora vaikutus ohjelman ymmärrettävyyteen ja ylläpidettävyyteen. Ohjelman toiminnan ymmärtäminen ei edellytä niin paljon muistettavia asioita eikä se ole henkisesti niin kuormittavaa. Muutosten tekeminen on helpompaa, koska ohjelman ymmärtää helpommin ja muutettavan kohdan löytäminen on nopeampaa.

Tekniseen yksinkertaistumiseen vaikutti useampi asia, joista uudistetun sovelluksen sisäisen rakenteiden ja tietorakenteiden yksinkertaistaminen olivat oleellisimpia asioita. Ohjelmointikielen vaihtamisella funktionaaliseen Clojureen oli myös vaikutusta ohjelmakoodin pienenemiseen, mutta arkkitehtuuritason muutoksilla oli suuremmat vaikutukset. Clojuren selkeimpänä hyötynä oli koodin nopea tuottaminen ja toisten tekemän koodin nopeampi ymmärtäminen sen tiiviimmän esitysmuodon takia. Ohjelmiston kehittämistä nopeutti mahdollisuus viedä koodimuutokset ajonaikaisesti suorituksessa olevaan sovellukseen.

Kokonaisarkkitehtuurin kannalta sovelluksen irrottaminen monoliittiarkkitehtuurista itsenäiseksi ei vaikuttanut juurikaan tekniseen yksinkertaistumiseen, mutta tämä on jatkokehityksen ja ylläpidettävyyden kannalta merkittävä asia. Päivitykset voidaan tehdä itsenäisesti eikä koko järjestelmää tarvitse päivittää kerrallaan tuotantoympäristöön. Kehittäjien näkökulmasta tarkasteltuna pienemmän itsenäisen osan kehittäminen on nopeampaa, koska kaikkia koodeja ei tarvitse kääntää ja suorittaa. Myös kehitysympäristöt

toimivat paremmin pienemmän sovelluksen kanssa. Uudistetun ohjelman elinkaaren kannalta teknologioiden modernisointi oli kannattavaa ja se luo jatkoaikaa ohjelman käyttämiselle.

Vanhan ja uudistetun sovelluksen vertaaminen puhtaasti tekniseltä kannalta ei ole helppoa. Ohjelmointiparadigmat ovat erilaiset, minkä vaikutuksena osa perinteisistä metriikoista on heikosti sellaisenaan soveltuvia tai ainakin niitä pitää muokata paremmin soveltuviksi. Lisäksi teknisten mittareiden osalta voi saada virheellisen kuvan sovelluksesta. Vanha viisaus – sitä saa, mitä mittaa – pätee tässäkin asiassa ja onnistuneita asioita on mahdollista korostaa ja vähemmän onnistuneita piilottaa mittareiden näkymättömiin. Valittujen metriikoiden lisäksi sovellusohjelmia voitaisiin mitata myös muilla metriikoilla, joiden avulla arviota voi tarkentaa vielä entisestään. Näiden teknisten metriikoiden lisäksi huomattava osa arvioinnista perustui haastatteluihin.

Kyselyn ja haastattelun perusteella voi havaita, että yksittäisin osa-alueittain tarkasteltuna uudistus oli onnistunut. Aliluvussa 5.4 esitetyn taulukon 4 mukaisen jaottelun mukaan tarkasteltuna tehokkuus ja suorituskyky parantuivat huomattavasti. Laatu ei juurikaan parantunut, koska myös vanha järjestelmä koettiin laadukkaaksi. Toisaalta uuden järjestelmän laatua ei arvioitu huonommaksi kuin vanhan järjestelmän. Uudistettu sovellus sisälsi toimivia ja uudelleenkäytettäviä konsepteja, mutta kokonaisuus on keskittynyt hyvin ominaispiirteiseen toimialaan eikä uudelleenkäyttöä nähdä sellaisenaan ilman suurempia muutoksia. Eniten uudelleenkäytettäviä asioita ovat toteutuksessa käytetyt periaatteet ja ideat. Uudistettu sovellus koettiin joustavammaksi ja jatkokehitettävämmäksi kuin alkuperäinen sovellus. Siihen on helpompi tehdä muutoksia ja uutta toiminnallisuutta. Jatkokehitettävyyteen vaikutti myös teknologioiden modernisointi, jolla vanhasta painolastista päästiin eroon.

Uudistus oli teknisesti tarkasteltuna erityisen hyvin onnistunut. Toiminnallisuuden ja ohjelmistoprojektin kannalta uudistus oli onnistunut, mutta lisäpanostusta olisi tarvinnut vielä enemmän alkuvaiheen suunnitteluun ja esiselvitykseen. Uudistettava sovellus vaikutti alkuvaiheessa yksinkertaisemmalta kuin se oikeasti olikaan. Tämä korostui erityisesti datan migraatiossa vanhasta järjestelmästä uuteen. Kokonaisuutta ajatellen uudistuksen voidaan sanoa olevan onnistunut.

8. YHTEENVETO JA POHDINTAA

Ohjelmistoissa tulee elinkaaren loppuvaiheessa lähes väistämättä eteen joko niiden käytön lopettaminen tai uudistaminen. Uudistamisen tavoitteena on usein vastata muuttuneisiin vaatimuksiin. Vaatimuksia tulee muun muassa toimintaympäristön muutoksista ja toisaalta toimintaympäristö voi toimia palautteena ohjelman toiminnasta eikä kaikkia vaatimuksia voi selvittää ennen kuin ohjelma on käytössä. Näin oli käynyt myös esimerkkitapauksessa.

Työssä tarkasteltiin, miten ohjelman uudistamiseen päädytään ja miten uudistaminen käytännössä kannattaa tehdä. Esimerkkitapauksessa tavoitteina olivat ylläpidon helpottaminen ja joustavuuden ja suorituskyvyn parantaminen. Näitä tavoitteita lähdettiin saavuttamaan irrottamalla osakokonaisuus omaksi itsenäisesti toimivaksi järjestelmäksi. Ohjelman sisäistä arkkitehtuuria muutettiin yksinkertaisemmaksi uusien vaatimusten ja toimintaympäristön palautteen perusteella. Ohjelman yksinkertaistamiseksi ohjelmointikieli vaihdettiin Javasta ilmaisuvoimaisempaan ja tiiviimpään Clojureen. Teknisessä mielessä uudistaminen oli erittäin onnistunut. Tavoite saavutettiin. Ohjelmakoodin, kooditiedostojen ja tietokantataulujen määrä väheni noin viidesosaan. Lisäksi ohjelmakoodin sisäinen rakenne yksinkertaistui huomattavasti.

Funktionaalisella paradigmalla on selkeitä hyötyjä ja se sopii hyvin web-järjestelmiin. Muuttumattomuus ja puhtaat funktiot helpottavat ohjelman ymmärtämistä ja vähentävät koodissa olevia ohjelmistovirheitä. Lisäksi se mahdollistaa rinnakkaistuvien ohjelmien tekemisen. Clojure valittiin korvaavaksi kieleksi sen funktionaalisen paradigman ja ajoympäristön perusteella. Sitä hyödynnettiin sekä käyttöliittymässä että palvelimen toteutuksessa unohtamatta yhteistä koodia, jota oli suoraviivaista käyttää molemmissa päissä. Ohjelmistokehittäjillä on yleisesti vahva osaaminen olio-ohjelmoinnista varsinkin verrattuna funktionaaliseen ohjelmointiin. Funktionaalinen paradigma voikin toimia mielen virkistäjänä ja sopivana vaihteluna.

Uudistusprojektit noudattavat normaalin ohjelmistoprojektin periaatteita. Siihen liittyvät samat vaiheet, riskit ja ominaispiirteet. Eroavaisuutena on projektin alkupuolella tapahtuva määrittelyn tekeminen. Uudistusprojekti perustuu vanhaan, joten vanhan sovelluksen tutkiminen ja analysointi ovat osa määrittelyä. Riippuen uudistuksen tasosta, tällä voi olla merkittäväkin rooli. Uudistusprojektissa on mahdollista panostaa esitutkimukseen ja tekniseen suunnitteluun. Mikäli nämä tehdään hyvin, uudistamista on helpompi seurata ja projektin valmistuminen on paremmin ennustettavissa. Kokonaistyömäärän

arviointi on helpompaa, mitä tarkemmin suunnitellaan askeleet tavoitteen saavuttamiseksi.

Ohjelman uudistamisen lisäksi eräs huomattava seikka on vanhan datan siirtäminen uuteen ympäristöön ja tietomalleihin. Tämä vie aikaa ja edellyttää vahvaa tuntemusta sekä vanhasta että uudesta sovelluksesta. Datamigraatioon kuluvaa aikaa ja työmäärää ei kannata jättää huomioimatta, sillä se voi olla yllättävän suuri.

Työssä tapaustutkimuksen kohteena olevan uudistamisen onnistuminen edellyttää ylläpitovaiheen seurantaa. Tämänhetkisen tilanteen ja arvion mukaan uudistus oli onnistunut, mutta esimerkiksi ylläpidon helpottamisen ja joustavuuden lisääntymisen todentaminen perustuu lopulta vielä tuntemattomien muuttuvasta toimintaympäristöstä aiheutuvien uusien ominaisuuksien tekemiseen.

Clojure soveltuu web-sovelluksiin ja runsaasti ohjelmallista datankäsittelyä sisältäviin ohjelmiin. Lisätutkimusta pitää tehdä esimerkiksi vertailemalla Clojurea muihin funktionaalisiin ohjelmointikieliin kuten Scalaan ja Groovyyn tai Javan uudempiin versioihin ja Kotliniin, joiden avulla voi myös ohjelmoida funktionaalisesti. Tuskin mikään näistä on paras kaikkiin tapauksiin, vaan eri ohjelmointikielillä on erilaisia vahvuuksia ja heikkouksia. Ohjelmointikielen tai ohjelmointiparadigman valinta ei kuitenkaan yksinään ratkaise ongelmia. Hyvällä ohjelmistosuunnittelulla ja sovelluksen laadulla on suuri merkitys.

LÄHTEET

[1] A. Abran, ISO 9126: Analysis of Quality Models and Measures, Software Metrics and Software Metrology, Wiley-IEEE Computer Society, 2010, s. 205-228

[2] A. Kaipiainen, Funktionaalinen ohjelmointi web-ohjelmistokehitysympäristössä, Diplomityö, Tampereen Teknillinen Yliopisto, 2016, Saatavilla: http://urn.fi/URN:NBN:fi:tty-201701021001, Viitattu 13.10.2019

[3] A. Miller, R. Hickey, Clojure spec Guide, Saatavilla: https://clojure.org/guides/spec, Viitattu 13.10.2019

[4] A. Miller, S. Halloway, Aaron Bedra, Programming Clojure, Third Edition, The Pragmatic Programmers, 2018

[5] About JavaScript, Mozilla, Saatavilla: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript, Viitattu 13.10.2019

[6] Angular, Saatavilla: https://angular.io, Viitattu 13.10.2019

[7] Apache JMeter, The Apache Software Foundation, Saatavilla: https://jmeter.apache.org/, Viitattu 13.10.2019

[8] Apache Tomcat, The Apache Software Foundation, Saatavilla: http://tomcat.apache.org/, Viitattu 18.10.2019

[9] Apache Wicket, Apache Software Foundation, Saatavilla: https://wicket.apache.org/, Viitattu 13.10.2019

[10] B. Kurniawan, Java 7: A Comprehensive Tutorial, Brainy Software Inc, First Edition, 2014, s. 2-10

[11] B. Larsen, Beginning HTML & CSS, Wiley, 2013

[12] C. Birchall, Re-engineering Legacy Software, Manning Publications, 2016

[13] D. Compton, R. Hickey, Clojure Reader Conditionals Guide, Rick Hickey, Saatavilla: https://clojure.org/guides/reader_conditionals, Viitattu 13.10.2019

[14] D. Simmonds, The Programming Paradigm Evolution, IEEE Computer Society, 2012, s. 93

[15] Developer Survey Results 2019, Stack Overflow, Saatavilla: https://insights.stackoverflow.com/survey/2019, Viitattu 13.10.2019

[16] E. Gat,Point of view: Lisp as an Alternative to Java, Intelligence, Vol. 11, 2000, s. 21-24

[17] Fielding, Roy, Architectural Styles and the Design of Network-based Software Architectures, 2000, Saatavilla: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, Viitattu 13.10.2019

[18] H. Sneed, C. Verhoef, Re-implementing a legacy system, The Jounal of Systems and Software, Vol. 155, 2019, s. 162-184

[19] HyperText Transfer Protocol – HTTP/1.1, IETF, 1999, Saatavilla: https://tools.ietf.org/html/rfc2616, Viitattu 13.10.2019

[20] I. Haikala, J. Märijärvi, Ohjelmistotuotanto, 10. painos, Talentum Media Oy, 2004, s. 204-211

[21] I. Haikala, T. Mikkonen, Ohjelmistotuotannon käytännöt, Talentum Oy, 2011

[22] I. Warren, J. Ransom, Renaissance: A Method to Support Software System Evolution, Proceedings of the 26th Annual International Computer Software and Applications Conference, IEEE, 2002

[23] IETF, RFC 6455 - The WebSocket Protocol, 2011, Saatavilla: https://tools.ietf.org/html/rfc6455, Viitattu: 13.10.2019

[24] ISO/IEC 25010, System and software quality models, International Organization for Standardization, International Electrotechnical Commission, 2011

[25] J. Anitha, M. Karthika, K. Alagarsamy, The Classification and Analysis of Risks in Reengineering System, International Journal of Computer Applications, Vol. 39, No. 18, 2012, s. 57-60, Saatavilla: https://pdfs.semanticscholar.org/175d/0b37d69ac854f00630f80d08bc97a41a03f3.pdf, Viitattu 13.10.2019

[26] J. McCarty, History of LISP, ACM SIGPLAN Notices, Vol. 13, Iss. 8, 1978, s. 217-223

[27] J. Peltomäki, Pieni Java 8 -kirja, BoD – Books on Demand, 2014

[28] Java 7 Release Highlights, Oracle, Saatavilla: https://java.com/en/download/faq/release7_changes.xml, Viitattu 13.10.2019

[29] K. Koskimies, Oliokirja, Satku, 2000, s. 23-24

[30] K. Koskimies, T. Mikkonen, Ohjelmistoarkkitehtuurit, Talentum Media Oy, 2005, s.136-138

[31] M. Abi-Antoun, J. Aldrich, W. Coelho, A case study in re-engineering to enforce architectural control flow and data sharing, The Journal of Systems and Software Vol. 80, Iss. 2, 2007, s. 240-264

[32] M. Gabrielli, S. Martini, Programming Languages: Principles and Paradigms, Springer, 2010

[33] M. Godfrey, D. German, On the evolution of Lehman's Laws, Journal of Software: Evolution and Process, Vol. 26, Iss. 7, 2013, s. 613-619

[34] M. Harsu, Ohjelmien ylläpito ja uudistaminen, Talentum, 2003

[35] M. Harsu, Ohjelmointikielet – Periaatteet, käsitteet, valintaperusteet, Talentum Media Oy, 2005

[36] M. Lehman, J. Ramil, P. Wernick, D.Perry, W. Turski, Metrics and laws of software evolution – The Nineties View, Proceedings Fourth International Software Metrics Symposium, IEEE, 1997, s. 21-22

[37] M. Lehman, Programs, Life Cycles, and Laws of Software Evolution, Proceedings of the IEEE, Vol. 68, No. 9, 1980, s.1061-1068

[38] M. McDonnell, Quick Clojure: Effective Functional Programming, Apress, 2017

[39] N. Rashid, M. Salam, R. ShahSani, F. Alam, Analysis of Risks in Re-Engineering Software Systems, International Journal of Computer Applications, Vol. 73, No. 11, 2013, s. 6-7

[40] Node.js, Node.js Foundation, Saatavilla: https://nodejs.org/, Viitattu 13.10.2019

[41] Octoverse 2018, Github Inc, Saatavilla: https://octoverse.github.com/projects#languages, Viitattu 13.10.2019

[42] OWASP, Saatavilla: https://www.owasp.org, Viitattu 13.10.2019

[43] P. Deitel, JavaScript™ for Programmers, Prentice Hall, 2009

[44] P. Eriksson, K. Koistinen, Monenlainen tapaustutkimus, Kuluttajatutkimuskeskuksen tuloksia ja selvityksiä, Kuluttajatutkimuskeskus, 2014, Saatavilla: https://helda.helsinki.fi/bitstream/handle/10138/153032/Tutkimuksia%20ja%20selvityksi%E4_11_2014_%20Monenlainen%20tapaustutkimus_Eriksson_Koistinen.pdf?sequence=1, Viitattu 13.10.2019

[45] R. Kazman, S. Woodsm J. Carriere, Requirements for Integrating Software Architecture and Reengineerin Models: CORUM II, IEEE, 1998

[46] R. Martin, Clean Architecture, Prentice Hall, 2018

[47] React – A JavaScript Library for building user interfaces, Saatavilla: https://reactjs.org, Viitattu 13.10.2019

[48] S. Blackheath, A. Jones, Functional Reactive Programming, Manning Publictions, 2016

[49] S. Casteleyn, F. Daniel, P. Dolog, M. Matera, Engineering Web Applications, Springer, 2009

[50] S. Ceri, Designing Data-Intensive Web Applications, Morgan Kaufmann, 2003

[51] S. Chidamber, C. Kamerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994

[52] S. Hirsijärvi, H. Hurme, Tutkimushaastattelu, Gaudeamus, 2015, s. 44-47

[53] T. Heikkilä, Tilastollinen tutkimus, Edita Publishing Oy, 2014, s.51

[54] T. McCabe, A measure of Complexity, IEEE Transactions on Software Engineering, Vol. 2, Iss. 4, 1976, s. 308-320

[55] T. Mens, S. Demeyer, Software Evolution, Springer, 2008

[56] TIOBE Index for June 2019, Saatavilla: https://www.tiobe.com/tiobe-index/, Viitattu 13.10.2019

[57] What is WildFly?, Red Hat Inc., Saatavilla: https://wildfly.org/about/, Viitattu 18.10.2019